

Tasks

- 1: [Install Tools](#)
 - 1.1: [Install and Set Up kubectl on Linux](#)
 - 1.2: [Install and Set Up kubectl on macOS](#)
 - 1.3: [Install and Set Up kubectl on Windows](#)
- 2: [Administer a Cluster](#)
 - 2.1: [Administration with kubeadm](#)
 - 2.1.1: [Certificate Management with kubeadm](#)
 - 2.1.2: [Configuring a cgroup driver](#)
 - 2.1.3: [Reconfiguring a kubeadm cluster](#)
 - 2.1.4: [Upgrading kubeadm clusters](#)
 - 2.1.5: [Upgrading Linux nodes](#)
 - 2.1.6: [Upgrading Windows nodes](#)
 - 2.1.7: [Changing The Kubernetes Package Repository](#)
 - 2.2: [Migrating from dockershim](#)
 - 2.2.1: [Changing the Container Runtime on a Node from Docker Engine to containerd](#)
 - 2.2.2: [Migrate Docker Engine nodes from dockershim to cri-dockerd](#)
 - 2.2.3: [Find Out What Container Runtime is Used on a Node](#)
 - 2.2.4: [Troubleshooting CNI plugin-related errors](#)
 - 2.2.5: [Check whether dockershim removal affects you](#)
 - 2.2.6: [Migrating telemetry and security agents from dockershim](#)
 - 2.3: [Generate Certificates Manually](#)
 - 2.4: [Manage Memory, CPU, and API Resources](#)
 - 2.4.1: [Configure Default Memory Requests and Limits for a Namespace](#)
 - 2.4.2: [Configure Default CPU Requests and Limits for a Namespace](#)
 - 2.4.3: [Configure Minimum and Maximum Memory Constraints for a Namespace](#)
 - 2.4.4: [Configure Minimum and Maximum CPU Constraints for a Namespace](#)
 - 2.4.5: [Configure Memory and CPU Quotas for a Namespace](#)
 - 2.4.6: [Configure a Pod Quota for a Namespace](#)
 - 2.5: [Install a Network Policy Provider](#)
 - 2.5.1: [Use Antrea for NetworkPolicy](#)
 - 2.5.2: [Use Calico for NetworkPolicy](#)
 - 2.5.3: [Use Cilium for NetworkPolicy](#)
 - 2.5.4: [Use Kube-router for NetworkPolicy](#)
 - 2.5.5: [Romana for NetworkPolicy](#)
 - 2.5.6: [Weave Net for NetworkPolicy](#)
 - 2.6: [Access Clusters Using the Kubernetes API](#)
 - 2.7: [Advertise Extended Resources for a Node](#)
 - 2.8: [Autoscale the DNS Service in a Cluster](#)
 - 2.9: [Change the default StorageClass](#)
 - 2.10: [Switching from Polling to CRI Event-based Updates to Container Status](#)
 - 2.11: [Change the Reclaim Policy of a PersistentVolume](#)

- 2.12: [Cloud Controller Manager Administration](#)
- 2.13: [Configure a kubelet image credential provider](#)
- 2.14: [Configure Quotas for API Objects](#)
- 2.15: [Control CPU Management Policies on the Node](#)
- 2.16: [Control Topology Management Policies on a node](#)
- 2.17: [Customizing DNS Service](#)
- 2.18: [Debugging DNS Resolution](#)
- 2.19: [Declare Network Policy](#)
- 2.20: [Developing Cloud Controller Manager](#)
- 2.21: [Enable Or Disable A Kubernetes API](#)
- 2.22: [Encrypting Confidential Data at Rest](#)
- 2.23: [Decrypt Confidential Data that is Already Encrypted at Rest](#)
- 2.24: [Guaranteed Scheduling For Critical Add-On Pods](#)
- 2.25: [IP Masquerade Agent User Guide](#)
- 2.26: [Limit Storage Consumption](#)
- 2.27: [Migrate Replicated Control Plane To Use Cloud Controller Manager](#)
- 2.28: [Namespaces Walkthrough](#)
- 2.29: [Operating etcd clusters for Kubernetes](#)
- 2.30: [Reserve Compute Resources for System Daemons](#)
- 2.31: [Running Kubernetes Node Components as a Non-root User](#)
- 2.32: [Safely Drain a Node](#)
- 2.33: [Securing a Cluster](#)
- 2.34: [Set Kubelet Parameters Via A Configuration File](#)
- 2.35: [Share a Cluster with Namespaces](#)
- 2.36: [Upgrade A Cluster](#)
- 2.37: [Use Cascading Deletion in a Cluster](#)
- 2.38: [Using a KMS provider for data encryption](#)
- 2.39: [Using CoreDNS for Service Discovery](#)
- 2.40: [Using NodeLocal DNSCache in Kubernetes Clusters](#)
- 2.41: [Using sysctls in a Kubernetes Cluster](#)
- 2.42: [Utilizing the NUMA-aware Memory Manager](#)
- 2.43: [Verify Signed Kubernetes Artifacts](#)
- 3: [Configure Pods and Containers](#)
 - 3.1: [Assign Memory Resources to Containers and Pods](#)
 - 3.2: [Assign CPU Resources to Containers and Pods](#)
 - 3.3: [Configure GMSA for Windows Pods and containers](#)
 - 3.4: [Resize CPU and Memory Resources assigned to Containers](#)
 - 3.5: [Configure RunAsUserName for Windows pods and containers](#)
 - 3.6: [Create a Windows HostProcess Pod](#)
 - 3.7: [Configure Quality of Service for Pods](#)
 - 3.8: [Assign Extended Resources to a Container](#)
 - 3.9: [Configure a Pod to Use a Volume for Storage](#)
 - 3.10: [Configure a Pod to Use a PersistentVolume for Storage](#)
 - 3.11: [Configure a Pod to Use a Projected Volume for Storage](#)
 - 3.12: [Configure a Security Context for a Pod or Container](#)
 - 3.13: [Configure Service Accounts for Pods](#)
 - 3.14: [Pull an Image from a Private Registry](#)
 - 3.15: [Configure Liveness, Readiness and Startup Probes](#)
 - 3.16: [Assign Pods to Nodes](#)
 - 3.17: [Assign Pods to Nodes using Node Affinity](#)
 - 3.18: [Configure Pod Initialization](#)

- 3.19: [Attach Handlers to Container Lifecycle Events](#)
- 3.20: [Configure a Pod to Use a ConfigMap](#)
- 3.21: [Share Process Namespace between Containers in a Pod](#)
- 3.22: [Use a User Namespace With a Pod](#)
- 3.23: [Create static Pods](#)
- 3.24: [Translate a Docker Compose File to Kubernetes Resources](#)
- 3.25: [Enforce Pod Security Standards by Configuring the Built-in Admission Controller](#)
- 3.26: [Enforce Pod Security Standards with Namespace Labels](#)
- 3.27: [Migrate from PodSecurityPolicy to the Built-In PodSecurity Admission Controller](#)
- 4: [Monitoring, Logging, and Debugging](#)
 - 4.1: [Troubleshooting Applications](#)
 - 4.1.1: [Debug Pods](#)
 - 4.1.2: [Debug Services](#)
 - 4.1.3: [Debug a StatefulSet](#)
 - 4.1.4: [Determine the Reason for Pod Failure](#)
 - 4.1.5: [Debug Init Containers](#)
 - 4.1.6: [Debug Running Pods](#)
 - 4.1.7: [Get a Shell to a Running Container](#)
 - 4.2: [Troubleshooting Clusters](#)
 - 4.2.1: [Troubleshooting kubectl](#)
 - 4.2.2: [Resource metrics pipeline](#)
 - 4.2.3: [Tools for Monitoring Resources](#)
 - 4.2.4: [Monitor Node Health](#)
 - 4.2.5: [Debugging Kubernetes nodes with cAdvisor](#)
 - 4.2.6: [Auditing](#)
 - 4.2.7: [Debugging Kubernetes Nodes With Kubectl](#)
 - 4.2.8: [Developing and debugging services locally using telepresence](#)
 - 4.2.9: [Windows debugging tips](#)
- 5: [Manage Kubernetes Objects](#)
 - 5.1: [Declarative Management of Kubernetes Objects Using Configuration Files](#)
 - 5.2: [Declarative Management of Kubernetes Objects Using Kustomize](#)
 - 5.3: [Managing Kubernetes Objects Using Imperative Commands](#)
 - 5.4: [Imperative Management of Kubernetes Objects Using Configuration Files](#)
 - 5.5: [Update API Objects in Place Using kubectl patch](#)
- 6: [Managing Secrets](#)
 - 6.1: [Managing Secrets using kubectl](#)
 - 6.2: [Managing Secrets using Configuration File](#)
 - 6.3: [Managing Secrets using Kustomize](#)
- 7: [Inject Data Into Applications](#)
 - 7.1: [Define a Command and Arguments for a Container](#)
 - 7.2: [Define Dependent Environment Variables](#)
 - 7.3: [Define Environment Variables for a Container](#)
 - 7.4: [Expose Pod Information to Containers Through Environment Variables](#)
 - 7.5: [Expose Pod Information to Containers Through Files](#)
 - 7.6: [Distribute Credentials Securely Using Secrets](#)
- 8: [Run Applications](#)
 - 8.1: [Run a Stateless Application Using a Deployment](#)
 - 8.2: [Run a Single-Instance Stateful Application](#)

- 8.3: [Run a Replicated Stateful Application](#)
- 8.4: [Scale a StatefulSet](#)
- 8.5: [Delete a StatefulSet](#)
- 8.6: [Force Delete StatefulSet Pods](#)
- 8.7: [Horizontal Pod Autoscaling](#)
- 8.8: [HorizontalPodAutoscaler Walkthrough](#)
- 8.9: [Specifying a Disruption Budget for your Application](#)
- 8.10: [Accessing the Kubernetes API from a Pod](#)
- 9: [Run Jobs](#)
 - 9.1: [Running Automated Tasks with a CronJob](#)
 - 9.2: [Coarse Parallel Processing Using a Work Queue](#)
 - 9.3: [Fine Parallel Processing Using a Work Queue](#)
 - 9.4: [Indexed Job for Parallel Processing with Static Work Assignment](#)
 - 9.5: [Job with Pod-to-Pod Communication](#)
 - 9.6: [Parallel Processing using Expansions](#)
 - 9.7: [Handling retriable and non-retriable pod failures with Pod failure policy](#)
- 10: [Access Applications in a Cluster](#)
 - 10.1: [Deploy and Access the Kubernetes Dashboard](#)
 - 10.2: [Accessing Clusters](#)
 - 10.3: [Configure Access to Multiple Clusters](#)
 - 10.4: [Use Port Forwarding to Access Applications in a Cluster](#)
 - 10.5: [Use a Service to Access an Application in a Cluster](#)
 - 10.6: [Connect a Frontend to a Backend Using Services](#)
 - 10.7: [Create an External Load Balancer](#)
 - 10.8: [List All Container Images Running in a Cluster](#)
 - 10.9: [Set up Ingress on Minikube with the NGINX Ingress Controller](#)
 - 10.10: [Communicate Between Containers in the Same Pod Using a Shared Volume](#)
 - 10.11: [Configure DNS for a Cluster](#)
 - 10.12: [Access Services Running on Clusters](#)
- 11: [Extend Kubernetes](#)
 - 11.1: [Configure the Aggregation Layer](#)
 - 11.2: [Use Custom Resources](#)
 - 11.2.1: [Extend the Kubernetes API with CustomResourceDefinitions](#)
 - 11.2.2: [Versions in CustomResourceDefinitions](#)
 - 11.3: [Set up an Extension API Server](#)
 - 11.4: [Configure Multiple Schedulers](#)
 - 11.5: [Use an HTTP Proxy to Access the Kubernetes API](#)
 - 11.6: [Use a SOCKS5 Proxy to Access the Kubernetes API](#)
 - 11.7: [Set up Konnectivity service](#)
- 12: [TLS](#)
 - 12.1: [Configure Certificate Rotation for the Kubelet](#)
 - 12.2: [Manage TLS Certificates in a Cluster](#)
 - 12.3: [Manual Rotation of CA Certificates](#)
- 13: [Manage Cluster Daemons](#)
 - 13.1: [Perform a Rolling Update on a DaemonSet](#)
 - 13.2: [Perform a Rollback on a DaemonSet](#)
 - 13.3: [Running Pods on Only Some Nodes](#)
- 14: [Networking](#)
 - 14.1: [Adding entries to Pod /etc/hosts with HostAliases](#)
 - 14.2: [Validate IPv4/IPv6 dual-stack](#)

- 15: [Extend kubectl with plugins](#)
- 16: [Manage HugePages](#)
- 17: [Schedule GPUs](#)

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.

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

1 - Install Tools

Set up Kubernetes tools on your computer.

kubectl

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

kubectl is installable on a variety of Linux platforms, macOS and Windows. Find your preferred operating system below.

- [Install kubectl on Linux](#)
- [Install kubectl on macOS](#)
- [Install kubectl on Windows](#)

kind

[kind](#) lets you run Kubernetes on your local computer. This tool requires that you have either [Docker](#) or [Podman](#) installed.

The kind [Quick Start](#) page shows you what you need to do to get up and running with kind.

[View kind Quick Start Guide](#)

minikube

Like `kind`, [minikube](#) is a tool that lets you run Kubernetes locally. `minikube` runs an all-in-one or a multi-node local Kubernetes cluster on your personal computer (including Windows, macOS and Linux PCs) so that you can try out Kubernetes, or for daily development work.

You can follow the official [Get Started!](#) guide if your focus is on getting the tool installed.

[View minikube Get Started! Guide](#)

Once you have `minikube` working, you can use it to [run a sample application](#).

kubeadm

You can use the [kubeadm](#) tool to create and manage Kubernetes clusters. It performs the actions necessary to get a minimum viable, secure cluster up and running in a user friendly way.

[Installing kubeadm](#) shows you how to install kubeadm. Once installed, you can use it to [create a cluster](#).

[View kubeadm Install Guide](#)

1.1 - Install and Set Up kubectl on Linux

Before you begin

You must use a kubectl version that is within one minor version difference of your cluster. For example, a v1.28 client can communicate with v1.27, v1.28, and v1.29 control planes. Using the latest compatible version of kubectl helps avoid unforeseen issues.

Install kubectl on Linux

The following methods exist for installing kubectl on Linux:

- [Install kubectl binary with curl on Linux](#)
- [Install using native package management](#)
- [Install using other package management](#)

Install kubectl binary with curl on Linux

1. Download the latest release with the command:



```
curl -LO "https://dl.k8s.io/release/$(curl -L -s https://dl.k8s.io/release/stable.txt)\" /usr/local/bin/kubectl
```

Note:

To download a specific version, replace the `$(curl -L -s https://dl.k8s.io/release/stable.txt)` portion of the command with the specific version.

For example, to download version 1.28.3 on Linux x86-64, type:



```
curl -LO https://dl.k8s.io/release/v1.28.3/bin/linux/amd64/kubectl
```

And for Linux ARM64, type:



```
curl -LO https://dl.k8s.io/release/v1.28.3/bin/linux/arm64/kubectl
```

2. Validate the binary (optional)

Download the kubectl checksum file:



```
curl -LO https://dl.k8s.io/.../checksums/.../v1.28.3/kubectl.sha256
```

```
curl -LO "https://dl.k8s.io/release/$(curl -L -s https://dl.k8s.io/release/stable.txt)" --output kube
```

Validate the kubectl binary against the checksum file:

```
echo "$(cat kubectl.sha256) kubectl" | sha256sum --check
```

If valid, the output is:

```
kubectl: OK
```

If the check fails, sha256 exits with nonzero status and prints output similar to:

```
kubectl: FAILED  
sha256sum: WARNING: 1 computed checksum did NOT match
```

Note: Download the same version of the binary and checksum.

3. Install kubectl

```
sudo install -o root -g root -m 0755 kubectl /usr/local/bin/kubectl
```

Note:

If you do not have root access on the target system, you can still install kubectl to the `~/.local/bin` directory:

```
chmod +x kubectl  
mkdir -p ~/.local/bin  
mv ./kubectl ~/.local/bin/kubectl  
# and then append (or prepend) ~/.Local/bin to $PATH
```

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

```
kubectl version --client
```

Or use this for detailed view of version:

```
kubectl version --client --output=yaml
```

Install using native package management

[Debian-based distributions](#)

[Red Hat-based distributions](#)

SUSE-based distributions

1. Update the `apt` package index and install packages needed to use the Kubernetes `apt` repository:

```
sudo apt-get update  
# apt-transport-https may be a dummy package; if so, you can skip that package  
sudo apt-get install -y apt-transport-https ca-certificates curl
```

2. Download the public signing key for the Kubernetes package repositories. The same signing key is used for all repositories so you can disregard the version in the URL:

```
curl -fsSL https://pkgs.k8s.io/core:/stable:/v1.28/deb/Release.key | sudo gpg -
```

3. Add the appropriate Kubernetes `apt` repository. If you want to use Kubernetes version different than v1.28, replace v1.28 with the desired minor version in the command below:

```
# This overwrites any existing configuration in /etc/apt/sources.list.d/kubernetes.list  
echo 'deb [signed-by=/etc/apt/keyrings/kubernetes-apt-keyring.gpg] https://pkgs.k8s.io/core:/stable:/v1.28/deb/' | sudo tee /etc/apt/sources.list.d/kubernetes.list
```

Note: To upgrade `kubectl` to another minor release, you'll need to bump the version in `/etc/apt/sources.list.d/kubernetes.list` before running `apt-get update` and `apt-get upgrade`. This procedure is described in more detail in [Changing The Kubernetes Package Repository](#).

4. Update `apt` package index, then install `kubectl`:

```
sudo apt-get update  
sudo apt-get install -y kubectl
```

Note: In releases older than Debian 12 and Ubuntu 22.04, `/etc/apt/keyrings` does not exist by default, and can be created using `sudo mkdir -m 755 /etc/apt/keyrings`

Install using other package management

[Snap](#) [Homebrew](#)

If you are on Ubuntu or another Linux distribution that supports the [snap](#) package manager, `kubectl` is available as a [snap](#) application.

```
snap install kubectl --classic  
kubectl version --client
```

Verify `kubectl` configuration

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. By default, `kubectl` configuration is located at `~/.kube/config`.

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 configured correctly or is not able to connect to a Kubernetes cluster.

```
The connection to the server <server-name:port> was refused - did you s
```

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
```

Optional kubectl configurations and plugins

Enable shell completion

kubectl provides completion support for Bash, Zsh, Fish, and PowerShell, which can save you a lot of typing.

Below are the procedures to set up completion for Bash, Fish, and Zsh.

[Bash](#) [Fish](#) [Zsh](#)

Introduction

The kubectl completion script for Bash can be generated with the command `kubectl completion bash`. Sourcing the completion script in your shell enables kubectl completion.

However, the completion script depends on [bash-completion](#), which means that you have to install this software first (you can test if you have bash-completion already installed by running `type _init_completion`).

Install bash-completion

bash-completion is provided by many package managers (see [here](#)). You can install it with `apt-get install bash-completion` OR `yum install bash-completion`, etc.

The above commands create `/usr/share/bash-completion/bash_completion`, which is the main script of bash-completion. Depending on your package manager, you have to manually source this file in your `~/.bashrc` file.

To find out, reload your shell and run `type _init_completion`. If the command succeeds, you're already set, otherwise add the following to your `~/.bashrc` file:

```
source /usr/share/bash-completion/bash_completion
```

Reload your shell and verify that bash-completion is correctly installed by typing `type _init_completion`.

Enable kubectl autocompletion

Bash

You now need to ensure that the kubectl completion script gets sourced in all your shell sessions. There are two ways in which you can do this:

User [System](#)

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

If you have an alias for kubectl, you can extend shell completion to work with that alias:

```
echo 'alias k=kubectl' >>~/.bashrc
echo 'complete -o default -F __start_kubectl k' >>~/.bashrc
```

Note: bash-completion sources all completion scripts in [/etc/bash_completion.d](#).

Both approaches are equivalent. After reloading your shell, kubectl autocompletion should be working. To enable bash autocompletion in current session of shell, source the `~/.bashrc` file:

```
source ~/.bashrc
```

Install `kubectl convert` plugin

A plugin for Kubernetes command-line tool `kubectl`, which allows you to convert manifests between different API versions. This can be particularly helpful to migrate manifests to a non-deprecated api version with newer Kubernetes release. For more info, visit [migrate to non deprecated apis](#)

1. Download the latest release with the command:

x86-64 [ARM64](#)

```
curl -LO "https://dl.k8s.io/release/$(curl -L -s https://dl.k8s.io/release/stable.txt)" --output kubectlConvert
```

2. Validate the binary (optional)

Download the `kubectl-convert` checksum file:

x86-64 [ARM64](#)

```
curl -LO "https://dl.k8s.io/release/$(curl -L -s https://dl.k8s.io/release/stable.txt)" --output kubectlConvert
```

Validate the `kubectl-convert` binary against the checksum file:

```
echo "$(cat kubectl-convert.sha256) kubectl-convert" | sha256sum
```

If valid, the output is:

```
kubectl-convert: OK
```

If the check fails, `sha256` exits with nonzero status and prints output similar to:

```
kubectl-convert: FAILED
sha256sum: WARNING: 1 computed checksum did NOT match
```

Note: Download the same version of the binary and checksum.

3. Install `kubectl-convert`

```
sudo install -o root -g root -m 0755 kubectl-convert /usr/local/b
```

4. Verify plugin is successfully installed

```
kubectl convert --help
```

If you do not see an error, it means the plugin is successfully installed.

5. After installing the plugin, clean up the installation files:

```
rm kubectl-convert kubectl-convert.sha256
```

What's next

- [Install Minikube](#)
- See the [getting started guides](#) for more about creating clusters.
- [Learn how to launch and expose your application.](#)
- If you need access to a cluster you didn't create, see the [Sharing Cluster Access document](#).
- Read the [kubectl reference docs](#)

1.2 - Install and Set Up kubectl on macOS

Before you begin

You must use a kubectl version that is within one minor version difference of your cluster. For example, a v1.28 client can communicate with v1.27, v1.28, and v1.29 control planes. Using the latest compatible version of kubectl helps avoid unforeseen issues.

Install kubectl on macOS

The following methods exist for installing kubectl on macOS:

- [Install kubectl on macOS](#)
 - [Install kubectl binary with curl on macOS](#)
 - [Install with Homebrew on macOS](#)
 - [Install with Macports on macOS](#)
- [Verify kubectl configuration](#)
- [Optional kubectl configurations and plugins](#)
 - [Enable shell autocompletion](#)
 - [Install kubectl_convert plugin](#)

Install kubectl binary with curl on macOS

1. Download the latest release:



A screenshot of a web browser window. At the top, there are two tabs: "Intel" and "Apple Silicon". The "Apple Silicon" tab is currently active. Below the tabs, there is a large text input field containing a command-line instruction: "curl -LO \"https://dl.k8s.io/release/\$(curl -L -s https://dl.k8s.io/release/stable.txt)\"". At the bottom of the browser window, there is a horizontal scrollbar with arrows on both ends.

Note:

To download a specific version, replace the `$(curl -L -s https://dl.k8s.io/release/stable.txt)` portion of the command with the specific version.

For example, to download version 1.28.3 on Intel macOS, type:

```
curl -LO "https://dl.k8s.io/release/v1.28.3/bin/darwin/amd64/"
```

And for macOS on Apple Silicon, type:

```
curl -LO "https://dl.k8s.io/release/v1.28.3/bin/darwin/arm64/"
```

2. Validate the binary (optional)

Download the kubectl checksum file:

[Intel](#) [Apple Silicon](#)

```
curl -LO "https://dl.k8s.io/release/$(curl -L -s https://dl.k8s.io/release/stable.txt)" --output kubectl
```

Validate the kubectl binary against the checksum file:

```
echo "$(cat kubectl.sha256)  kubectl" | shasum -a 256 --check
```

If valid, the output is:

```
kubectl: OK
```

If the check fails, `shasum` exits with nonzero status and prints output similar to:

```
kubectl: FAILED
shasum: WARNING: 1 computed checksum did NOT match
```

Note: Download the same version of the binary and checksum.

3. Make the kubectl binary executable.

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

4. Move the kubectl binary to a file location on your system `PATH`.

```
sudo mv ./kubectl /usr/local/bin/kubectl
sudo chown root: /usr/local/bin/kubectl
```

Note: Make sure `/usr/local/bin` is in your `PATH` environment variable.

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

```
kubectl version --client
```

Or use this for detailed view of version:

```
kubectl version --client --output=yaml
```

6. After installing the plugin, clean up the installation files:

```
rm kubectl kubectl.sha256
```

Install with Homebrew on macOS

If you are on macOS and using [Homebrew](#) package manager, you can install kubectl with Homebrew.

1. Run the installation command:

```
brew install kubectl
```

or

```
brew install kubernetes-cli
```

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

```
kubectl version --client
```

Install with Macports on macOS

If you are on macOS and using [Macports](#) package manager, you can install kubectl with Macports.

1. Run the installation command:

```
sudo port selfupdate  
sudo port install kubectl
```

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

```
kubectl version --client
```

Verify kubectl configuration

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. By default, kubectl configuration is located at `~/.kube/config`.

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 configured correctly or is not able to connect to a Kubernetes cluster.

```
The connection to the server <server-name:port> was refused - did you s
```

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
```

Optional kubectl configurations and plugins

Enable shell completion

kubectl provides completion support for Bash, Zsh, Fish, and PowerShell which can save you a lot of typing.

Below are the procedures to set up completion for Bash, Fish, and Zsh.

[Bash](#) [Fish](#) [Zsh](#)

Introduction

The kubectl completion script for Bash can be generated with `kubectl completion bash`. Sourcing this script in your shell enables kubectl completion.

However, the kubectl completion script depends on [bash-completion](#) which you thus have to previously install.

Warning: There are two versions of bash-completion, v1 and v2. V1 is for Bash 3.2 (which is the default on macOS), and v2 is for Bash 4.1+. The kubectl completion script **doesn't work** correctly with bash-completion v1 and Bash 3.2. It requires **bash-completion v2** and **Bash 4.1+**. Thus, to be able to correctly use kubectl completion on macOS, you have to install and use Bash 4.1+ ([instructions](#)). The following instructions assume that you use Bash 4.1+ (that is, any Bash version of 4.1 or newer).

Upgrade Bash

The instructions here assume you use Bash 4.1+. You can check your Bash's version by running:

```
echo $BASH_VERSION
```

If it is too old, you can install/upgrade it using Homebrew:

```
brew install bash
```

Reload your shell and verify that the desired version is being used:

```
echo $BASH_VERSION $SHELL
```

Homebrew usually installs it at `/usr/local/bin/bash`.

Install bash-completion

Note: As mentioned, these instructions assume you use Bash 4.1+, which means you will install bash-completion v2 (in contrast to Bash 3.2 and bash-completion v1, in which case kubectl completion won't work).

You can test if you have bash-completion v2 already installed with `type _init_completion`. If not, you can install it with Homebrew:

```
brew install bash-completion@2
```

As stated in the output of this command, add the following to your `~/.bash_profile` file:

```
brew_etc="$(brew --prefix)/etc" && [[ -r "${brew_etc}/profile.d/bash_completion.sh"
```

Reload your shell and verify that bash-completion v2 is correctly installed with `type _init_completion`.

Enable kubectl autocompletion

You now have to ensure that the kubectl completion script gets sourced in all your shell sessions. There are multiple ways to achieve this:

- Source the completion script in your `~/.bash_profile` file:

```
echo 'source <(kubectl completion bash)' >>~/.bash_profile
```

- Add the completion script to the `/usr/local/etc/bash_completion.d` directory:

```
kubectl completion bash >/usr/local/etc/bash_completion.d/kubectl
```

- If you have an alias for kubectl, you can extend shell completion to work with that alias:

```
echo 'alias k=kubectl' >>~/.bash_profile
echo 'complete -o default -F __start_kubectl k' >>~/.bash_profile
```

- If you installed kubectl with Homebrew (as explained [here](#)), then the kubectl completion script should already be in `/usr/local/etc/bash_completion.d/kubectl`. In that case, you don't need to do anything.

Note: The Homebrew installation of bash-completion v2 sources all the files in the `BASH_COMPLETION_COMPAT_DIR` directory, that's why the latter two methods work.

In any case, after reloading your shell, kubectl completion should be working.

Install `kubectl convert` plugin

A plugin for Kubernetes command-line tool `kubectl`, which allows you to convert manifests between different API versions. This can be particularly helpful to migrate manifests to a non-deprecated api version with newer Kubernetes release. For more info, visit [migrate to non deprecated apis](#)

1. Download the latest release with the command:

Intel

[Apple Silicon](#)

```
curl -LO "https://dl.k8s.io/release/$(curl -L -s https://dl.k8s.io/release/stable.txt)" --output kubelet
```

2. Validate the binary (optional)

Download the kubectl-convert checksum file:

[Intel](#)

[Apple Silicon](#)

```
curl -LO "https://dl.k8s.io/release/$(curl -L -s https://dl.k8s.io/release/stable.txt)" --output kubelet
```

Validate the kubectl-convert binary against the checksum file:

```
echo "$(cat kubectl-convert.sha256)  kubectl-convert" | shasum -a
```

If valid, the output is:

```
kubectl-convert: OK
```

If the check fails, `shasum` exits with nonzero status and prints output similar to:

```
kubectl-convert: FAILED
shasum: WARNING: 1 computed checksum did NOT match
```

Note: Download the same version of the binary and checksum.

3. Make kubectl-convert binary executable

```
chmod +x ./kubectl-convert
```

4. Move the kubectl-convert binary to a file location on your system

PATH .

```
sudo mv ./kubectl-convert /usr/local/bin/kubectl-convert
sudo chown root: /usr/local/bin/kubectl-convert
```

Note: Make sure `/usr/local/bin` is in your PATH environment variable.

5. Verify plugin is successfully installed

```
kubectl convert --help
```

If you do not see an error, it means the plugin is successfully installed.

6. After installing the plugin, clean up the installation files:

```
rm kubectl-convert kubectl-convert.sha256
```

Uninstall kubectl on macOS

Depending on how you installed `kubectl`, use one of the following methods.

Uninstall kubectl using the command-line

1. Locate the `kubectl` binary on your system:

```
which kubectl
```

2. Remove the `kubectl` binary:

```
sudo rm <path>
```

Replace `<path>` with the path to the `kubectl` binary from the previous step. For example, `sudo rm /usr/local/bin/kubectl`.

Uninstall kubectl using homebrew

If you installed `kubectl` using Homebrew, run the following command:

```
brew remove kubectl
```

What's next

- [Install Minikube](#)
- See the [getting started guides](#) for more about creating clusters.
- [Learn how to launch and expose your application.](#)
- If you need access to a cluster you didn't create, see the [Sharing Cluster Access document](#).
- Read the [kubectl reference docs](#)

1.3 - Install and Set Up kubectl on Windows

Before you begin

You must use a kubectl version that is within one minor version difference of your cluster. For example, a v1.28 client can communicate with v1.27, v1.28, and v1.29 control planes. Using the latest compatible version of kubectl helps avoid unforeseen issues.

Install kubectl on Windows

The following methods exist for installing kubectl on Windows:

- [Install kubectl binary with curl on Windows](#)
- [Install on Windows using Chocolatey, Scoop, or winget](#)

Install kubectl binary with curl on Windows

1. Download the latest 1.28 patch release: [kubectl 1.28.3](#).

Or if you have `curl` installed, use this command:

```
curl.exe -LO "https://dl.k8s.io/release/v1.28.3/bin/windows/amd64/kubectl"
```

Note: To find out the latest stable version (for example, for scripting), take a look at <https://dl.k8s.io/release/stable.txt>.

2. Validate the binary (optional)

Download the `kubectl` checksum file:

```
curl.exe -LO "https://dl.k8s.io/v1.28.3/bin/windows/amd64/kubectl.sha256"
```

Validate the `kubectl` binary against the checksum file:

- Using Command Prompt to manually compare `CertUtil`'s output to the checksum file downloaded:

```
CertUtil -hashfile kubectl.exe SHA256  
type kubectl.exe.sha256
```

- Using PowerShell to automate the verification using the `-eq` operator to get a `True` OR `False` result:

```
$([Get-FileHash] -Algorithm SHA256 .\kubectl.exe).Hash -eq $(
```

3. Append or prepend the `kubectl` binary folder to your `PATH` environment variable.
4. Test to ensure the version of `kubectl` is the same as downloaded:

```
kubectl version --client
```

Or use this for detailed view of version:

```
kubectl version --client --output=yaml
```

Note: [Docker Desktop for Windows](#) adds its own version of `kubectl` to `PATH`. If you have installed Docker Desktop before, you may need to place your `PATH` entry before the one added by the Docker Desktop installer or remove the Docker Desktop's `kubectl`.

Install on Windows using Chocolatey, Scoop, or winget

1. To install `kubectl` on Windows you can use either [Chocolatey](#) package manager, [Scoop](#) command-line installer, or [winget](#) package manager.

[choco](#) [scoop](#) [winget](#)

```
choco install kubernetes-cli
```

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

```
kubectl version --client
```

3. Navigate to your home directory:

```
# If you're using cmd.exe, run: cd %USERPROFILE%
cd ~
```

4. Create the `.kube` directory:

```
mkdir .kube
```

5. Change to the `.kube` directory you just created:

```
cd .kube
```

6. Configure `kubectl` to use a remote Kubernetes cluster:

```
New-Item config -type file
```

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

Verify kubectl configuration

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. By default, kubectl configuration is located at `~/.kube/config`.

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 configured correctly or is not able to connect to a Kubernetes cluster.

```
The connection to the server <server-name:port> was refused - did you s
```

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
```

Optional kubectl configurations and plugins

Enable shell completion

kubectl provides completion support for Bash, Zsh, Fish, and PowerShell, which can save you a lot of typing.

Below are the procedures to set up completion for PowerShell.

The kubectl completion script for PowerShell can be generated with the command `kubectl completion powershell`.

To do so in all your shell sessions, add the following line to your `$PROFILE` file:

```
kubectl completion powershell | Out-String | Invoke-Expression
```

This command will regenerate the auto-completion script on every PowerShell start up. You can also add the generated script directly to your `$PROFILE` file.

To add the generated script to your `$PROFILE` file, run the following line in your powershell prompt:

```
kubectl completion powershell >> $PROFILE
```

After reloading your shell, kubectl autocomplete should be working.

Install `kubectl convert` plugin

A plugin for Kubernetes command-line tool `kubectl`, which allows you to convert manifests between different API versions. This can be particularly helpful to migrate manifests to a non-deprecated api version with newer Kubernetes release. For more info, visit [migrate to non deprecated apis](#)

1. Download the latest release with the command:

```
curl.exe -LO "https://dl.k8s.io/release/v1.28.3/bin/windows/amd64/kubectlConvert.exe"
```

2. Validate the binary (optional).

Download the `kubectl-convert` checksum file:

```
curl.exe -LO "https://dl.k8s.io/v1.28.3/bin/windows/amd64/kubectlConvert.exe.sha256"
```

Validate the `kubectl-convert` binary against the checksum file:

- Using Command Prompt to manually compare `CertUtil`'s output to the checksum file downloaded:

```
CertUtil -hashfile kubectl-convert.exe SHA256  
type kubectl-convert.exe.sha256
```

- Using PowerShell to automate the verification using the `-eq` operator to get a `True` or `False` result:

```
$($("CertUtil -hashfile .\kubectl-convert.exe SHA256")[1] -rep
```

3. Append or prepend the `kubectl-convert` binary folder to your `PATH` environment variable.

4. Verify the plugin is successfully installed.

```
kubectl convert --help
```

If you do not see an error, it means the plugin is successfully installed.

5. After installing the plugin, clean up the installation files:

```
del kubectl-convert.exe kubectl-convert.exe.sha256
```

What's next

- [Install Minikube](#)
- See the [getting started guides](#) for more about creating clusters.
- [Learn how to launch and expose your application.](#)
- If you need access to a cluster you didn't create, see the [Sharing Cluster Access document](#).
- Read the [kubectl reference docs](#)

2 - Administer a Cluster

Learn common tasks for administering a cluster.

2.1 - Administration with kubeadm

2.1.1 - Certificate Management with kubeadm

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

Client certificates generated by [kubeadm](#) expire after 1 year. This page explains how to manage certificate renewals with kubeadm. It also covers other tasks related to kubeadm certificate management.

Before you begin

You should be familiar with [PKI certificates and requirements in Kubernetes](#).

Using custom certificates

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

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

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

External CA mode

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

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

[PKI certificates and requirements](#) includes guidance on setting up a cluster to use an external CA.

Check certificate expiration

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

```
kubeadm certs check-expiration
```

The output is similar to this:

CERTIFICATE	EXPIRES	RESIDUAL TIME	CEI
admin.conf	Dec 30, 2020 23:36 UTC	364d	
apiserver	Dec 30, 2020 23:36 UTC	364d	ca
apiserver-etcd-client	Dec 30, 2020 23:36 UTC	364d	etcd
apiserver-kubelet-client	Dec 30, 2020 23:36 UTC	364d	ca
controller-manager.conf	Dec 30, 2020 23:36 UTC	364d	
etcd-healthcheck-client	Dec 30, 2020 23:36 UTC	364d	etcd
etcd-peer	Dec 30, 2020 23:36 UTC	364d	etcd
etcd-server	Dec 30, 2020 23:36 UTC	364d	etcd
front-proxy-client	Dec 30, 2020 23:36 UTC	364d	front
scheduler.conf	Dec 30, 2020 23:36 UTC	364d	
CERTIFICATE AUTHORITY	EXPIRES	RESIDUAL TIME	EXTERNAL
ca	Dec 28, 2029 23:36 UTC	9y	no
etcd-ca	Dec 28, 2029 23:36 UTC	9y	no
front-proxy-ca	Dec 28, 2029 23:36 UTC	9y	no

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

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

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

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

Warning:

On nodes created with `kubeadm init`, prior to kubeadm version 1.17, there is a [bug](#) where you manually have to modify the contents of `kubelet.conf`. After `kubeadm init` finishes, you should update `kubelet.conf` to point to the rotated kubelet client certificates, by replacing `client-certificate-data` and `client-key-data` with:

```
client-certificate: /var/lib/kubelet/pki/kubelet-client-current.pem
client-key: /var/lib/kubelet/pki/kubelet-client-current.pem
```

Automatic certificate renewal

kubeadm renews all the certificates during control plane [upgrade](#).

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

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

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

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

Manual certificate renewal

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

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

After running the command you should restart the control plane Pods. This is required since dynamic certificate reload is currently not supported for all components and certificates. [Static Pods](#) are managed by the local kubelet and not by the API Server, thus kubectl cannot be used to delete and restart them. To restart a static Pod you can temporarily remove its manifest file from `/etc/kubernetes/manifests/` and wait for 20 seconds (see the `fileCheckFrequency` value in [KubeletConfiguration struct](#)). The kubelet will terminate the Pod if it's no longer in the manifest directory. You can then move the file back and after another `fileCheckFrequency` period, the kubelet will recreate the Pod and the certificate renewal for the component can complete.

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

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

`kubeadm certs renew` can renew any specific certificate or, with the subcommand `all`, it can renew all of them, as shown below:

```
kubeadm certs renew all
```

Note:

Clusters built with kubeadm often copy the `admin.conf` certificate into `$HOME/.kube/config`, as instructed in [Creating a cluster with kubeadm](#). On such a system, to update the contents of `$HOME/.kube/config` after

renewing the `admin.conf`, you must run the following commands:

```
sudo cp -i /etc/kubernetes/admin.conf $HOME/.kube/config
sudo chown $(id -u):$(id -g) $HOME/.kube/config
```

Renew certificates with the Kubernetes certificates API

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

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

Set up a signer

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

The built-in signer is part of [kube-controller-manager](#).

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

If you're creating a new cluster, you can use a kubeadm [configuration file](#):

```
apiVersion: kubeadm.k8s.io/v1beta3
kind: ClusterConfiguration
controllerManager:
  extraArgs:
    cluster-signing-cert-file: /etc/kubernetes/pki/ca.crt
    cluster-signing-key-file: /etc/kubernetes/pki/ca.key
```

Create certificate signing requests (CSR)

See [Create CertificateSigningRequest](#) for creating CSRs with the Kubernetes API.

Renew certificates with external CA

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

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

Create certificate signing requests (CSR)

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

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

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

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

- In `openssl` this is done with the [openssl ca command](#).
- In `cfssl` you specify [usages in the config file](#).

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

Certificate authority (CA) rotation

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

For more information about manual rotation or replacement of CA, see [manual rotation of CA certificates](#).

Enabling signed kubelet serving certificates

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

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

```
apiVersion: kubeadm.k8s.io/v1beta3
kind: ClusterConfiguration
---
apiVersion: kubelet.config.k8s.io/v1beta1
kind: KubeletConfiguration
serverTLSBootstrap: true
```

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

- Find and edit the `kubelet-config-1.28` ConfigMap in the `kube-system` namespace. In that ConfigMap, the `kubelet` key has a [KubeletConfiguration](#) document as its value. Edit the `KubeletConfiguration` document to set `serverTLSBootstrap: true`.
- On each node, add the `serverTLSBootstrap: true` field in `/var/lib/kubelet/config.yaml` and restart the kubelet with `systemctl restart kubelet`

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

These CSRs can be viewed using:

```
kubectl get csr
```

NAME	AGE	SIGNERNAME	REQUESTOR
csr-9wvgt	112s	kubernetes.io/kubelet-serving	system:node:work
csr-lz97v	1m58s	kubernetes.io/kubelet-serving	system:node:cont

To approve them you can do the following:

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

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

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

Note: This section links to third party projects that provide functionality required by Kubernetes. The Kubernetes project authors aren't responsible for these projects, which are listed alphabetically. To add a project to this list, read the [content guide](#) before submitting a change. [More information](#).

Third party custom controllers can be used:

- [kubelet-csr-approver](#)

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

Generating kubeconfig files for additional users

During cluster creation, kubeadm signs the certificate in the `admin.conf` to have `Subject: O = system:masters, CN = kubernetes-admin`. `system:masters` is a break-glass, super user group that bypasses the authorization layer (for example, [RBAC](#)). Sharing the `admin.conf` with additional users is **not recommended!**

Instead, you can use the `kubeadm kubeconfig user` command to generate kubeconfig files for additional users. The command accepts a mixture of command line flags and [kubeadm configuration](#) options. The generated kubeconfig will be written to stdout and can be piped to a file using

```
kubeadm kubeconfig user ... > somefile.conf .
```

Example configuration file that can be used with `--config`:

```
# example.yaml
apiVersion: kubeadm.k8s.io/v1beta3
kind: ClusterConfiguration
# Will be used as the target "cluster" in the kubeconfig
clusterName: "kubernetes"
# Will be used as the "server" (IP or DNS name) of this cluster in the
controlPlaneEndpoint: "some-dns-address:6443"
# The cluster CA key and certificate will be loaded from this local directory
certificatesDir: "/etc/kubernetes/pki"
```

Make sure that these settings match the desired target cluster settings. To see the settings of an existing cluster use:

```
kubectl get cm kubeadm-config -n kube-system -o=jsonpath="{.data.ClusterConfiguration}"
```

The following example will generate a kubeconfig file with credentials valid for 24 hours for a new user `johndoe` that is part of the `appdevs` group:

```
kubeadm kubeconfig user --config example.yaml --org appdevs --client-name johndoe --validity 24h
```

The following example will generate a kubeconfig file with administrator credentials valid for 1 week:

```
kubeadm kubeconfig user --config example.yaml --client-name admin --validity 1w
```

2.1.2 - Configuring a cgroup driver

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

Before you begin

You should be familiar with the Kubernetes [container runtime requirements](#).

Configuring the container runtime cgroup driver

The [Container runtimes](#) page explains that the `systemd` driver is recommended for kubeadm based setups instead of the kubelet's `default cgroupfs` driver, because kubeadm manages the kubelet as a [systemd service](#).

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

Configuring the kubelet cgroup driver

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

Note:

In v1.22 and later, if the user does not set the `cgroupDriver` field under `KubeletConfiguration`, kubeadm defaults it to `systemd`.

In Kubernetes v1.28, you can enable automatic detection of the cgroup driver as an alpha feature. See [systemd cgroup driver](#) for more details.

A minimal example of configuring the field explicitly:

```
# kubeadm-config.yaml
kind: ClusterConfiguration
apiVersion: kubeadm.k8s.io/v1beta3
kubernetesVersion: v1.21.0
---
kind: KubeletConfiguration
apiVersion: kubelet.config.k8s.io/v1beta1
cgroupDriver: systemd
```

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

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

Note:

Kubeadm uses the same `KubeletConfiguration` for all nodes in the cluster. The `KubeletConfiguration` is stored in a [ConfigMap](#) object under the `kube-system` namespace.

Executing the sub commands `init`, `join` and `upgrade` would result in kubeadm writing the `KubeletConfiguration` as a file under `/var/lib/kubelet/config.yaml` and passing it to the local node kubelet.

Using the `cgroupfs` driver

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

See the below section on "[Modify the kubelet ConfigMap](#)" for details on how to be explicit about the value.

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

Migrating to the `systemd` driver

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

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

Modify the kubelet ConfigMap

- Call `kubectl edit cm kubelet-config -n kube-system`.
- Either modify the existing `cgroupDriver` value or add a new field that looks like this:

```
cgroupDriver: systemd
```

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

Update the cgroup driver on all nodes

For each node in the cluster:

- [Drain the node](#) using `kubectl drain <node-name> --ignore-daemonsets`
- Stop the kubelet using `systemctl stop kubelet`
- Stop the container runtime
- Modify the container runtime cgroup driver to `systemd`
- Set `cgroupDriver: systemd` in `/var/lib/kubelet/config.yaml`

- Start the container runtime
- Start the kubelet using `systemctl start kubelet`
- [Uncordon the node](#) using `kubectl uncordon <node-name>`

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

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

2.1.3 - Reconfiguring a kubeadm cluster

kubeadm does not support automated ways of reconfiguring components that were deployed on managed nodes. One way of automating this would be by using a custom [operator](#).

To modify the components configuration you must manually edit associated cluster objects and files on disk.

This guide shows the correct sequence of steps that need to be performed to achieve kubeadm cluster reconfiguration.

Before you begin

- You need a cluster that was deployed using kubeadm
- Have administrator credentials (`/etc/kubernetes/admin.conf`) and network connectivity to a running kube-apiserver in the cluster from a host that has kubectl installed
- Have a text editor installed on all hosts

Reconfiguring the cluster

kubeadm writes a set of cluster wide component configuration options in ConfigMaps and other objects. These objects must be manually edited. The command `kubectl edit` can be used for that.

The `kubectl edit` command will open a text editor where you can edit and save the object directly.

You can use the environment variables `KUBECONFIG` and `KUBE_EDITOR` to specify the location of the kubectl consumed kubeconfig file and preferred text editor.

For example:

```
KUBECONFIG=/etc/kubernetes/admin.conf KUBE_EDITOR=nano kubectl edit <pa
```

Note: Upon saving any changes to these cluster objects, components running on nodes may not be automatically updated. The steps below instruct you on how to perform that manually.

Warning: Component configuration in ConfigMaps is stored as unstructured data (YAML string). This means that validation will not be performed upon updating the contents of a ConfigMap. You have to be careful to follow the documented API format for a particular component configuration and avoid introducing typos and YAML indentation mistakes.

Applying cluster configuration changes

Updating the [ClusterConfiguration](#)

During cluster creation and upgrade, kubeadm writes its [ClusterConfiguration](#) in a ConfigMap called `kubeadm-config` in the `kube-system` namespace.

To change a particular option in the `ClusterConfiguration` you can edit the `ConfigMap` with this command:

```
kubectl edit cm -n kube-system kubeadm-config
```

The configuration is located under the `data.ClusterConfiguration` key.

Note: The `ClusterConfiguration` includes a variety of options that affect the configuration of individual components such as `kube-apiserver`, `kube-scheduler`, `kube-controller-manager`, `CoreDNS`, `etcd` and `kube-proxy`. Changes to the configuration must be reflected on node components manually.

Reflecting `ClusterConfiguration` changes on control plane nodes

`kubeadm` manages the control plane components as static Pod manifests located in the directory `/etc/kubernetes/manifests`. Any changes to the `ClusterConfiguration` under the `apiServer`, `controllerManager`, `scheduler` or `etcd` keys must be reflected in the associated files in the `manifests` directory on a control plane node.

Such changes may include:

- `extraArgs` - requires updating the list of flags passed to a component container
- `extraMounts` - requires updated the volume mounts for a component container
- `*SANs` - requires writing new certificates with updated Subject Alternative Names.

Before proceeding with these changes, make sure you have backed up the directory `/etc/kubernetes/`.

To write new certificates you can use:

```
kubeadm init phase certs <component-name> --config <config-file>
```

To write new manifest files in `/etc/kubernetes/manifests` you can use:

```
kubeadm init phase control-plane <component-name> --config <config-file>
```

The `<config-file>` contents must match the updated `ClusterConfiguration`. The `<component-name>` value must be the name of the component.

Note: Updating a file in `/etc/kubernetes/manifests` will tell the `kubelet` to restart the static Pod for the corresponding component. Try doing these changes one node at a time to leave the cluster without downtime.

Applying kubelet configuration changes

Updating the `KubeletConfiguration`

During cluster creation and upgrade, kubeadm writes its [KubeletConfiguration](#) in a ConfigMap called `kubelet-config` in the `kube-system` namespace.

You can edit the ConfigMap with this command:

```
kubectl edit cm -n kube-system kubelet-config
```

The configuration is located under the `data.kubelet` key.

Reflecting the kubelet changes

To reflect the change on kubeadm nodes you must do the following:

- Log in to a kubeadm node
- Run `kubeadm upgrade node phase kubelet-config` to download the latest `kubelet-config` ConfigMap contents into the local file `/var/lib/kubelet/config.yaml`
- Edit the file `/var/lib/kubelet/kubeadm-flags.env` to apply additional configuration with flags
- Restart the kubelet service with `systemctl restart kubelet`

Note: Do these changes one node at a time to allow workloads to be rescheduled properly.

Note: During `kubeadm upgrade`, kubeadm downloads the `KubeletConfiguration` from the `kubelet-config` ConfigMap and overwrite the contents of `/var/lib/kubelet/config.yaml`. This means that node local configuration must be applied either by flags in `/var/lib/kubelet/kubeadm-flags.env` or by manually updating the contents of `/var/lib/kubelet/config.yaml` after `kubeadm upgrade`, and then restarting the kubelet.

Applying kube-proxy configuration changes

Updating the `KubeProxyConfiguration`

During cluster creation and upgrade, kubeadm writes its `KubeProxyConfiguration` in a ConfigMap in the `kube-system` namespace called `kube-proxy`.

This ConfigMap is used by the `kube-proxy` DaemonSet in the `kube-system` namespace.

To change a particular option in the `KubeProxyConfiguration`, you can edit the ConfigMap with this command:

```
kubectl edit cm -n kube-system kube-proxy
```

The configuration is located under the `data.config.conf` key.

Reflecting the kube-proxy changes

Once the `kube-proxy` ConfigMap is updated, you can restart all kube-proxy Pods:

Obtain the Pod names:

```
kubectl get po -n kube-system | grep kube-proxy
```

Delete a Pod with:

```
kubectl delete po -n kube-system <pod-name>
```

New Pods that use the updated ConfigMap will be created.

Note: Because kubeadm deploys kube-proxy as a DaemonSet, node specific configuration is unsupported.

Applying CoreDNS configuration changes

Updating the CoreDNS Deployment and Service

kubeadm deploys CoreDNS as a Deployment called `coredns` and with a Service `kube-dns`, both in the `kube-system` namespace.

To update any of the CoreDNS settings, you can edit the Deployment and Service objects:

```
kubectl edit deployment -n kube-system coredns  
kubectl edit service -n kube-system kube-dns
```

Reflecting the CoreDNS changes

Once the CoreDNS changes are applied you can delete the CoreDNS Pods:

Obtain the Pod names:

```
kubectl get po -n kube-system | grep coredns
```

Delete a Pod with:

```
kubectl delete po -n kube-system <pod-name>
```

New Pods with the updated CoreDNS configuration will be created.

Note: kubeadm does not allow CoreDNS configuration during cluster creation and upgrade. This means that if you execute `kubeadm upgrade apply`, your changes to the CoreDNS objects will be lost and must be reapplied.

Persisting the reconfiguration

During the execution of `kubeadm upgrade` on a managed node, kubeadm might overwrite configuration that was applied after the cluster was created (reconfiguration).

Persisting Node object reconfiguration

kubeadm writes Labels, Taints, CRI socket and other information on the Node object for a particular Kubernetes node. To change any of the contents of this Node object you can use:

```
kubectl edit no <node-name>
```

During `kubeadm upgrade` the contents of such a Node might get overwritten. If you would like to persist your modifications to the Node object after upgrade, you can prepare a [kubectl patch](#) and apply it to the Node object:

```
kubectl patch no <node-name> --patch-file <patch-file>
```

Persisting control plane component reconfiguration

The main source of control plane configuration is the `ClusterConfiguration` object stored in the cluster. To extend the static Pod manifests configuration, [patches](#) can be used.

These patch files must remain as files on the control plane nodes to ensure that they can be used by the `kubeadm upgrade ... --patches <directory>`.

If reconfiguration is done to the `ClusterConfiguration` and static Pod manifests on disk, the set of node specific patches must be updated accordingly.

Persisting kubelet reconfiguration

Any changes to the `KubeletConfiguration` stored in `/var/lib/kubelet/config.yaml` will be overwritten on `kubeadm upgrade` by downloading the contents of the cluster wide `kubelet-config ConfigMap`. To persist kubelet node specific configuration either the file `/var/lib/kubelet/config.yaml` has to be updated manually post-upgrade or the file `/var/lib/kubelet/kubeadm-flags.env` can include flags. The kubelet flags override the associated `KubeletConfiguration` options, but note that some of the flags are deprecated.

A kubelet restart will be required after changing `/var/lib/kubelet/config.yaml` or `/var/lib/kubelet/kubeadm-flags.env`.

What's next

- [Upgrading kubeadm clusters](#)
- [Customizing components with the kubeadm API](#)
- [Certificate management with kubeadm](#)
- [Find more about kubeadm set-up](#)

2.1.4 - Upgrading kubeadm clusters

This page explains how to upgrade a Kubernetes cluster created with kubeadm from version 1.27.x to version 1.28.x, and from version 1.28.x to 1.28.y (where $y > x$). Skipping MINOR versions when upgrading is unsupported. For more details, please visit [Version Skew Policy](#).

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

- [Upgrading a kubeadm cluster from 1.26 to 1.27](#)
- [Upgrading a kubeadm cluster from 1.25 to 1.26](#)
- [Upgrading a kubeadm cluster from 1.24 to 1.25](#)
- [Upgrading a kubeadm cluster from 1.23 to 1.24](#)

The upgrade workflow at high level is the following:

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

Before you begin

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

Additional information

- The instructions below outline when to drain each node during the upgrade process. If you are performing a **minor** version upgrade for any kubelet, you **must** first drain the node (or nodes) that you are upgrading. In the case of control plane nodes, they could be running CoreDNS Pods or other critical workloads. For more information see [Draining nodes](#).
- All containers are restarted after upgrade, because the container spec hash value is changed.
- To verify that the kubelet service has successfully restarted after the kubelet has been upgraded, you can execute `systemctl status kubelet` or view the service logs with `journalctl -xeu kubelet`.
- Usage of the `--config` flag of `kubeadm upgrade` with [kubeadm configuration API types](#) with the purpose of reconfiguring the cluster is not recommended and can have unexpected results. Follow the steps in [Reconfiguring a kubeadm cluster](#) instead.

Changing the package repository

If you're using the community-owned package repositories (`pkgs.k8s.io`), you need to enable the package repository for the desired Kubernetes minor release. This is explained in [Changing the Kubernetes package repository](#) document.

Note: The legacy package repositories (`apt.kubernetes.io` and `yum.kubernetes.io`) have been [deprecated and frozen starting from September 13, 2023](#). [Using the new package repositories hosted at `pkgs.k8s.io`](#) is strongly recommended and required in order to install Kubernetes versions released after September 13, 2023.

The deprecated legacy repositories, and their contents, might be removed at any time in the future and without a further notice period. The new package repositories provide downloads for Kubernetes versions starting with v1.24.0.

Determine which version to upgrade to

Find the latest patch release for Kubernetes 1.28 using the OS package manager:

[Ubuntu, Debian or HypriotOS](#)

[CentOS, RHEL or Fedora](#)

```
# Find the latest 1.28 version in the list.  
# It should look like 1.28.x-*, where x is the latest patch.  
apt update  
apt-cache madison kubeadm
```

Upgrading control plane nodes

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

Call "kubeadm upgrade"

For the first control plane node

1. Upgrade kubeadm:

[Ubuntu, Debian or HypriotOS](#)

[CentOS, RHEL or Fedora](#)

```
# replace x in 1.28.x-* with the latest patch version  
apt-mark unhold kubeadm && \  
apt-get update && apt-get install -y kubeadm='1.28.x-*' && \  
apt-mark hold kubeadm
```

2. Verify that the download works and has the expected version:

```
kubeadm version
```

3. Verify the upgrade plan:

```
kubeadm upgrade plan
```

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

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

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

4. Choose a version to upgrade to, and run the appropriate command.

For example:

```
# replace x with the patch version you picked for this upgrade
sudo kubeadm upgrade apply v1.28.x
```

Once the command finishes you should see:

```
[upgrade/successful] SUCCESS! Your cluster was upgraded to "v1.28"
[upgrade/kubelet] Now that your control plane is upgraded, please
```

Note: For versions earlier than v1.28, kubeadm defaulted to a mode that upgrades the addons (including CoreDNS and kube-proxy) immediately during `kubeadm upgrade apply`, regardless of whether there are other control plane instances that have not been upgraded. This may cause compatibility problems. Since v1.28, kubeadm defaults to a mode that checks whether all the control plane instances have been upgraded before starting to upgrade the addons. You must perform control plane instances upgrade sequentially or at least ensure that the last control plane instance upgrade is not started until all the other control plane instances have been upgraded completely, and the addons upgrade will be performed after the last control plane instance is upgraded. If you want to keep the old upgrade behavior, please enable the `UpgradeAddonsBeforeControlPlane` feature gate by `kubeadm upgrade apply --feature-gates=UpgradeAddonsBeforeControlPlane=true`. The Kubernetes project does not in general recommend enabling this feature gate, you should instead change your upgrade process or cluster addons so that you do not need to enable the legacy behavior. The `UpgradeAddonsBeforeControlPlane` feature gate will be removed in a future release.

5. Manually upgrade your CNI provider plugin.

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

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

For the other control plane nodes

Same as the first control plane node but use:

```
sudo kubeadm upgrade node
```

instead of:

```
sudo kubeadm upgrade apply
```

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

Drain the node

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

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

Upgrade kubelet and kubectl

1. Upgrade the kubelet and kubectl:

[Ubuntu, Debian or HypriotOS](#)

[CentOS, RHEL or Fedora](#)

```
# replace x in 1.28.x-* with the latest patch version
apt-mark unhold kubelet kubectl && \
apt-get update && apt-get install -y kubelet='1.28.x-*' kubectl
apt-mark hold kubelet kubectl
```

2. Restart the kubelet:

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

Uncordon the node

Bring the node back online by marking it schedulable:

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

Upgrade worker nodes

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

The following pages show how to upgrade Linux and Windows worker nodes:

- [Upgrade Linux nodes](#)
- [Upgrade Windows nodes](#)

Verify the status of the cluster

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

```
kubectl get nodes
```

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

Recovering from a failure state

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

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

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

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

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

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

How it works

`kubeadm upgrade apply` does the following:

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

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

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

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

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

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

2.1.5 - Upgrading Linux nodes

This page explains how to upgrade a Linux Worker Nodes created with kubeadm.

Before you begin

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

- [Killercoda](#)
- [Play with Kubernetes](#)

To check the version, enter `kubectl version`.

- Familiarize yourself with [the process for upgrading the rest of your kubeadm cluster](#). You will want to upgrade the control plane nodes before upgrading your Linux Worker nodes.

Changing the package repository

If you're using the community-owned package repositories (`pkgs.k8s.io`), you need to enable the package repository for the desired Kubernetes minor release. This is explained in [Changing the Kubernetes package repository](#) document.

Note: The legacy package repositories (`apt.kubernetes.io` and `yum.kubernetes.io`) have been [deprecated and frozen starting from September 13, 2023](#). **Using the new package repositories hosted at `pkgs.k8s.io` is strongly recommended and required in order to install Kubernetes versions released after September 13, 2023.** The deprecated legacy repositories, and their contents, might be removed at any time in the future and without a further notice period. The new package repositories provide downloads for Kubernetes versions starting with v1.24.0.

Upgrading worker nodes

Upgrade kubeadm

Upgrade kubeadm:

[Ubuntu, Debian or HypriotOS](#)

[CentOS, RHEL or Fedora](#)

```
# replace x in 1.28.x-* with the latest patch version
apt-mark unhold kubeadm && \
apt-get update && apt-get install -y kubeadm='1.28.x-*' && \
apt-mark hold kubeadm
```

Call "kubeadm upgrade"

For worker nodes this upgrades the local kubelet configuration:

```
sudo kubeadm upgrade node
```

Drain the node

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

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

Upgrade kubelet and kubectl

1. Upgrade the kubelet and kubectl:

[Ubuntu, Debian or HypriotOS](#)

[CentOS, RHEL or Fedora](#)

```
# replace x in 1.28.x-* with the latest patch version
apt-mark unhold kubelet kubectl && \
apt-get update && apt-get install -y kubelet='1.28.x-*' kubed
apt-mark hold kubelet kubectl
```

2. Restart the kubelet:

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

Uncordon the node

Bring the node back online by marking it schedulable:

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

What's next

- See how to [Upgrade Windows nodes](#).

2.1.6 - Upgrading Windows nodes

FEATURE STATE: [Kubernetes v1.18 \[beta\]](#)

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

Before you begin

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

- [Killercoda](#)
- [Play with Kubernetes](#)

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

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

Upgrading worker nodes

Upgrade kubeadm

1. From the Windows node, upgrade kubeadm:

```
# replace 1.28.3 with your desired version
curl.exe -Lo <path-to-kubeadm.exe> "https://dl.k8s.io/v1.28.3/bi
```

Drain the node

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

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

You should see output similar to this:

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

Upgrade the kubelet configuration

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

```
kubeadm upgrade node
```

Upgrade kubelet and kube-proxy

- From the Windows node, upgrade and restart the kubelet:

```
stop-service kubelet
curl.exe -Lo <path-to-kubelet.exe> "https://dl.k8s.io/v1.28.3/bin/windows/amd64/kubelet"
restart-service kubelet
```

- From the Windows node, upgrade and restart the kube-proxy.

```
stop-service kube-proxy
curl.exe -Lo <path-to-kube-proxy.exe> "https://dl.k8s.io/v1.28.3/bin/windows/amd64/kube-proxy"
restart-service kube-proxy
```

Note: If you are running kube-proxy in a HostProcess container within a Pod, and not as a Windows Service, you can upgrade kube-proxy by applying a newer version of your kube-proxy manifests.

Uncordon the node

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

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

What's next

- See how to [Upgrade Linux nodes](#).

2.1.7 - Changing The Kubernetes Package Repository

This page explains how to enable a package repository for a new Kubernetes minor release for users of the community-owned package repositories hosted at `pkgs.k8s.io`. Unlike the legacy package repositories, the community-owned package repositories are structured in a way that there's a dedicated package repository for each Kubernetes minor version.

Before you begin

This document assumes that you're already using the community-owned package repositories (`pkgs.k8s.io`). If that's not the case, it's strongly recommended to migrate to the community-owned package repositories as described in the [official announcement](#).

Note: The legacy package repositories (`apt.kubernetes.io` and `yum.kubernetes.io`) have been [deprecated and frozen starting from September 13, 2023](#). **Using the new package repositories hosted at `pkgs.k8s.io` is strongly recommended and required in order to install Kubernetes versions released after September 13, 2023.** The deprecated legacy repositories, and their contents, might be removed at any time in the future and without a further notice period. The new package repositories provide downloads for Kubernetes versions starting with v1.24.0.

Verifying if the Kubernetes package repositories are used

If you're unsure whether you're using the community-owned package repositories or the legacy package repositories, take the following steps to verify:

[Ubuntu, Debian or HypriotOS](#)

[CentOS, RHEL or Fedora](#)

[openSUSE or SLES](#)

Print the contents of the file that defines the Kubernetes `apt` repository:

```
# On your system, this configuration file could have a different name
pager /etc/apt/sources.list.d/kubernetes.list
```

If you see a line similar to:

```
deb [signed-by=/etc/apt/keyrings/kubernetes-apt-keyring.gpg] https://pkgs.k8s.io/core
```

You're using the Kubernetes package repositories and this guide applies to you.

Otherwise, it's strongly recommended to migrate to the Kubernetes package repositories as described in the [official announcement](#).

Note:

The URL used for the Kubernetes package repositories is not limited to `pkgs.k8s.io`, it can also be one of:

- `pkgs.k8s.io`

- pkgs.kubernetes.io
- packages.kubernetes.io

Switching to another Kubernetes package repository

This step should be done upon upgrading from one to another Kubernetes minor release in order to get access to the packages of the desired Kubernetes minor version.

[Ubuntu, Debian or HypriotOS](#)

[CentOS, RHEL or Fedora](#)

1. Open the file that defines the Kubernetes `apt` repository using a text editor of your choice:

```
nano /etc/apt/sources.list.d/kubernetes.list
```

You should see a single line with the URL that contains your current Kubernetes minor version. For example, if you're using v1.27, you should see this:

```
deb [signed-by=/etc/apt/keyrings/kubernetes-apt-keyring.gpg] https://pkgs.k8s.i
```

2. Change the version in the URL to **the next available minor release**, for example:

```
deb [signed-by=/etc/apt/keyrings/kubernetes-apt-keyring.gpg] https://pkgs.k8s.i
```

3. Save the file and exit your text editor. Continue following the relevant upgrade instructions.

What's next

- See how to [Upgrade Linux nodes](#).
- See how to [Upgrade Windows nodes](#).

2.2 - Migrating from dockershim

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

Since the announcement of [dockershim deprecation](#) in Kubernetes 1.20, there were questions on how this will affect various workloads and Kubernetes installations. Our [Dockershim Removal FAQ](#) is there to help you to understand the problem better.

Dockershim was removed from Kubernetes with the release of v1.24. If you use Docker Engine via dockershim as your container runtime and wish to upgrade to v1.24, it is recommended that you either migrate to another runtime or find an alternative means to obtain Docker Engine support. Check out the [container runtimes](#) section to know your options.

The version of Kubernetes with dockershim (1.23) is out of support and the v1.24 will run out of support [soon](#). Make sure to [report issues](#) you encountered with the migration so the issues can be fixed in a timely manner and your cluster would be ready for dockershim removal. After v1.24 running out of support, you will need to contact your Kubernetes provider for support or upgrade multiple versions at a time if there are critical issues affecting your cluster.

Your cluster might have more than one kind of node, although this is not a common configuration.

These tasks will help you to migrate:

- [Check whether Dockershim removal affects you](#)
- [Migrate Docker Engine nodes from dockershim to cri-dockerd](#)
- [Migrating telemetry and security agents from dockershim](#)

What's next

- Check out [container runtimes](#) to understand your options for an alternative.
- If you find a defect or other technical concern relating to migrating away from dockershim, you can [report an issue](#) to the Kubernetes project.

2.2.1 - Changing the Container Runtime on a Node from Docker Engine to containerd

This task outlines the steps needed to update your container runtime to containerd from Docker. It is applicable for cluster operators running Kubernetes 1.23 or earlier. This also covers an example scenario for migrating from dockershim to containerd. Alternative container runtimes can be picked from this [page](#).

Before you begin

Note: This section links to third party projects that provide functionality required by Kubernetes. The Kubernetes project authors aren't responsible for these projects, which are listed alphabetically. To add a project to this list, read the [content guide](#) before submitting a change. [More information.](#)

Install containerd. For more information see [containerd's installation documentation](#) and for specific prerequisite follow [the containerd guide](#).

Drain the node

```
kubectl drain <node-to-drain> --ignore-daemonsets
```

Replace `<node-to-drain>` with the name of your node you are draining.

Stop the Docker daemon

```
systemctl stop kubelet  
systemctl disable docker.service --now
```

Install Containerd

Follow the [guide](#) for detailed steps to install containerd.

[Linux](#)

[Windows \(PowerShell\)](#)

1. Install the `containerd.io` package from the official Docker repositories. Instructions for setting up the Docker repository for your respective Linux distribution and installing the `containerd.io` package can be found at [Getting started with containerd](#).

2. Configure containerd:

```
sudo mkdir -p /etc/containerd  
containerd config default | sudo tee /etc/containerd/config.toml
```

3. Restart containerd:

```
sudo systemctl restart containerd
```

Configure the kubelet to use containerd as its container runtime

Edit the file `/var/lib/kubelet/kubeadm-flags.env` and add the containerd runtime to the flags; `--container-runtime=endpoint=unix:///run/containerd/containerd.sock`.

Users using kubeadm should be aware that the `kubeadm` tool stores the CRI socket for each host as an annotation in the Node object for that host. To change it you can execute the following command on a machine that has the `kubeadm /etc/kubernetes/admin.conf` file.

```
kubectl edit no <node-name>
```

This will start a text editor where you can edit the Node object. To choose a text editor you can set the `KUBE_EDITOR` environment variable.

- Change the value of `kubeadm.alpha.kubernetes.io/cri-socket` from `/var/run/dockershim.sock` to the CRI socket path of your choice (for example `unix:///run/containerd/containerd.sock`).

Note that new CRI socket paths must be prefixed with `unix://` ideally.

- Save the changes in the text editor, which will update the Node object.

Restart the kubelet

```
systemctl start kubelet
```

Verify that the node is healthy

Run `kubectl get nodes -o wide` and containerd appears as the runtime for the node we just changed.

Remove Docker Engine

Note: This section links to third party projects that provide functionality required by Kubernetes. The Kubernetes project authors aren't responsible for these projects, which are listed alphabetically. To add a project to this list, read the [content guide](#) before submitting a change. [More information.](#)

If the node appears healthy, remove Docker.

[CentOS](#) [Debian](#) [Fedora](#) [Ubuntu](#)

```
sudo yum remove docker-ce docker-ce-cli
```

The preceding commands don't remove images, containers, volumes, or customized configuration files on your host. To delete them, follow Docker's instructions to [Uninstall Docker Engine](#).

Caution: Docker's instructions for uninstalling Docker Engine create a risk of deleting containerd. Be careful when executing commands.

Uncordon the node

```
kubectl uncordon <node-to-uncordon>
```

Replace `<node-to-uncordon>` with the name of your node you previously drained.

2.2.2 - Migrate Docker Engine nodes from dockershim to cri-dockerd

Note: This section links to third party projects that provide functionality required by Kubernetes. The Kubernetes project authors aren't responsible for these projects, which are listed alphabetically. To add a project to this list, read the [content guide](#) before submitting a change. [More information.](#)

This page shows you how to migrate your Docker Engine nodes to use `cri-dockerd` instead of `dockershim`. You should follow these steps in these scenarios:

- You want to switch away from `dockershim` and still use Docker Engine to run containers in Kubernetes.
- You want to upgrade to Kubernetes v1.28 and your existing cluster relies on `dockershim`, in which case you must migrate from `dockershim` and `cri-dockerd` is one of your options.

To learn more about the removal of `dockershim`, read the [FAQ page](#).

What is cri-dockerd?

In Kubernetes 1.23 and earlier, you could use Docker Engine with Kubernetes, relying on a built-in component of Kubernetes named `dockershim`. The `dockershim` component was removed in the Kubernetes 1.24 release; however, a third-party replacement, `cri-dockerd`, is available. The `cri-dockerd` adapter lets you use Docker Engine through the Container Runtime Interface.

Note: If you already use `cri-dockerd`, you aren't affected by the `dockershim` removal. Before you begin, [Check whether your nodes use the dockershim](#).

If you want to migrate to `cri-dockerd` so that you can continue using Docker Engine as your container runtime, you should do the following for each affected node:

1. Install `cri-dockerd`.
2. Cordon and drain the node.
3. Configure the kubelet to use `cri-dockerd`.
4. Restart the kubelet.
5. Verify that the node is healthy.

Test the migration on non-critical nodes first.

You should perform the following steps for each node that you want to migrate to `cri-dockerd`.

Before you begin

- `cri-dockerd` installed and started on each node.
- A [network plugin](#).

Cordon and drain the node

1. Cordon the node to stop new Pods scheduling on it:

```
kubectl cordon <NODE_NAME>
```

Replace `<NODE_NAME>` with the name of the node.

2. Drain the node to safely evict running Pods:

```
kubectl drain <NODE_NAME> \
--ignore-daemonsets
```

Configure the kubelet to use cri-dockerd

The following steps apply to clusters set up using the kubeadm tool. If you use a different tool, you should modify the kubelet using the configuration instructions for that tool.

1. Open `/var/lib/kubelet/kubeadm-flags.env` on each affected node.
2. Modify the `--container-runtime-endpoint` flag to `unix:///var/run/cri-dockerd.sock`.
3. Modify the `--container-runtime` flag to `remote` (unavailable in Kubernetes v1.27 and later).

The kubeadm tool stores the node's socket as an annotation on the `Node` object in the control plane. To modify this socket for each affected node:

1. Edit the YAML representation of the `Node` object:

```
KUBECONFIG=/path/to/admin.conf kubectl edit no <NODE_NAME>
```

Replace the following:

- o `/path/to/admin.conf` : the path to the kubectl configuration file, `admin.conf`.
 - o `<NODE_NAME>` : the name of the node you want to modify.
2. Change `kubeadm.alpha.kubernetes.io/cri-socket` from `/var/run/dockershim.sock` to `unix:///var/run/cri-dockerd.sock`.
 3. Save the changes. The `Node` object is updated on save.

Restart the kubelet

```
systemctl restart kubelet
```

Verify that the node is healthy

To check whether the node uses the `cri-dockerd` endpoint, follow the instructions in [Find out which runtime you use](#). The `--container-runtime-endpoint` flag for the kubelet should be `unix:///var/run/cri-dockerd.sock`.

Uncordon the node

Uncordon the node to let Pods schedule on it:

```
kubectl uncordon <NODE_NAME>
```

What's next

- Read the [dockershim removal FAQ](#).
- [Learn how to migrate from Docker Engine with dockershim to containerd](#).

2.2.3 - Find Out What Container Runtime is Used on a Node

This page outlines steps to find out what [container runtime](#) the nodes in your cluster use.

Depending on the way you run your cluster, the container runtime for the nodes may have been pre-configured or you need to configure it. If you're using a managed Kubernetes service, there might be vendor-specific ways to check what container runtime is configured for the nodes. The method described on this page should work whenever the execution of `kubectl` is allowed.

Before you begin

Install and configure `kubectl`. See [Install Tools](#) section for details.

Find out the container runtime used on a Node

Use `kubectl` to fetch and show node information:

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

The output is similar to the following. The column `CONTAINER-RUNTIME` outputs the runtime and its version.

For Docker Engine, the output is similar to this:

NAME	STATUS	VERSION	CONTAINER-RUNTIME
node-1	Ready	v1.16.15	docker://19.3.1
node-2	Ready	v1.16.15	docker://19.3.1
node-3	Ready	v1.16.15	docker://19.3.1

If your runtime shows as Docker Engine, you still might not be affected by the removal of dockershim in Kubernetes v1.24. [Check the runtime endpoint](#) to see if you use dockershim. If you don't use dockershim, you aren't affected.

For containerd, the output is similar to this:

NAME	STATUS	VERSION	CONTAINER-RUNTIME
node-1	Ready	v1.19.6	containerd://1.4.1
node-2	Ready	v1.19.6	containerd://1.4.1
node-3	Ready	v1.19.6	containerd://1.4.1

Find out more information about container runtimes on [Container Runtimes](#) page.

Find out what container runtime endpoint you use

The container runtime talks to the kubelet over a Unix socket using the [CRI protocol](#), which is based on the gRPC framework. The kubelet acts as a client, and the runtime acts as the server. In some cases, you might find it useful to know which socket your nodes use. For example, with the removal of dockershim in Kubernetes v1.24 and later, you might want to know whether you use Docker Engine with dockershim.

Note: If you currently use Docker Engine in your nodes with `cri-dockerd`, you aren't affected by the dockershim removal.

You can check which socket you use by checking the kubelet configuration on your nodes.

1. Read the starting commands for the kubelet process:

```
tr '\0' ' ' < /proc/"$(pgrep kubelet)"/cmdline
```

If you don't have `tr` or `pgrep`, check the command line for the kubelet process manually.

2. In the output, look for the `--container-runtime` flag and the `--container-runtime-endpoint` flag.

- If your nodes use Kubernetes v1.23 and earlier and these flags aren't present or if the `--container-runtime` flag is not `remote`, you use the dockershim socket with Docker Engine. The `--container-runtime` command line argument is not available in Kubernetes v1.27 and later.
- If the `--container-runtime-endpoint` flag is present, check the socket name to find out which runtime you use. For example, `unix:///run/containerd/containerd.sock` is the containerd endpoint.

If you want to change the Container Runtime on a Node from Docker Engine to containerd, you can find out more information on [migrating from Docker Engine to containerd](#), or, if you want to continue using Docker Engine in Kubernetes v1.24 and later, migrate to a CRI-compatible adapter like [cri-dockerd](#).

2.2.4 - Troubleshooting CNI plugin-related errors

To avoid CNI plugin-related errors, verify that you are using or upgrading to a container runtime that has been tested to work correctly with your version of Kubernetes.

About the "Incompatible CNI versions" and "Failed to destroy network for sandbox" errors

Service issues exist for pod CNI network setup and tear down in containerd v1.6.0-v1.6.3 when the CNI plugins have not been upgraded and/or the CNI config version is not declared in the CNI config files. The containerd team reports, "these issues are resolved in containerd v1.6.4."

With containerd v1.6.0-v1.6.3, if you do not upgrade the CNI plugins and/or declare the CNI config version, you might encounter the following "Incompatible CNI versions" or "Failed to destroy network for sandbox" error conditions.

Incompatible CNI versions error

If the version of your CNI plugin does not correctly match the plugin version in the config because the config version is later than the plugin version, the containerd log will likely show an error message on startup of a pod similar to:

```
incompatible CNI versions; config is \"1.0.0\", plugin supports [\"0.1.
```

To fix this issue, [update your CNI plugins and CNI config files](#).

Failed to destroy network for sandbox error

If the version of the plugin is missing in the CNI plugin config, the pod may run. However, stopping the pod generates an error similar to:

```
ERRO[2022-04-26T00:43:24.518165483Z] StopPodSandbox for "b" failed  
error="failed to destroy network for sandbox \"bbc85f891eaf060c5a879e27
```

This error leaves the pod in the not-ready state with a network namespace still attached. To recover from this problem, [edit the CNI config file](#) to add the missing version information. The next attempt to stop the pod should be successful.

Updating your CNI plugins and CNI config files

If you're using containerd v1.6.0-v1.6.3 and encountered "Incompatible CNI versions" or "Failed to destroy network for sandbox" errors, consider updating your CNI plugins and editing the CNI config files.

Here's an overview of the typical steps for each node:

1. [Safely drain and cordon the node](#).
2. After stopping your container runtime and kubelet services, perform the following upgrade operations:

- If you're running CNI plugins, upgrade them to the latest version.
- If you're using non-CNI plugins, replace them with CNI plugins. Use the latest version of the plugins.
- Update the plugin configuration file to specify or match a version of the CNI specification that the plugin supports, as shown in the following ["An example containerd configuration file"](#) section.
- For `containerd`, ensure that you have installed the latest version (v1.0.0 or later) of the CNI loopback plugin.
- Upgrade node components (for example, the `kubelet`) to Kubernetes v1.24
- Upgrade to or install the most current version of the container runtime.

3. Bring the node back into your cluster by restarting your container runtime and `kubelet`. Uncordon the node (`kubectl uncordon <nodename>`).

An example containerd configuration file

The following example shows a configuration for `containerd` runtime v1.6.x, which supports a recent version of the CNI specification (v1.0.0).

Please see the documentation from your plugin and networking provider for further instructions on configuring your system.

On Kubernetes, `containerd` runtime adds a loopback interface, `lo`, to pods as a default behavior. The `containerd` runtime configures the loopback interface via a CNI plugin, `loopback`. The `loopback` plugin is distributed as part of the `containerd` release packages that have the `cni` designation. `containerd` v1.6.0 and later includes a CNI v1.0.0-compatible loopback plugin as well as other default CNI plugins. The configuration for the loopback plugin is done internally by `containerd`, and is set to use CNI v1.0.0. This also means that the version of the `loopback` plugin must be v1.0.0 or later when this newer version `containerd` is started.

The following bash command generates an example CNI config. Here, the 1.0.0 value for the config version is assigned to the `cniVersion` field for use when `containerd` invokes the CNI bridge plugin.

```
cat << EOF | tee /etc/cni/net.d/10-containerd-net.conflist
{
  "cniVersion": "1.0.0",
  "name": "containerd-net",
  "plugins": [
    {
      "type": "bridge",
      "bridge": "cni0",
      "isGateway": true,
      "ipMasq": true,
      "promiscMode": true,
      "ipam": {
        "type": "host-local",
        "ranges": [
          [
            {
              "subnet": "10.88.0.0/16"
            },
            [
              {
                "subnet": "2001:db8:4860::/64"
              }
            ]
          ],
          "routes": [
            { "dst": "0.0.0.0/0" },
            { "dst": "::/0" }
          ]
        }
      },
      {
        "type": "portmap",
        "capabilities": {"portMappings": true},
        "externalSetMarkChain": "KUBE-MARK-MASQ"
      }
    ]
  }
EOF
```

Update the IP address ranges in the preceding example with ones that are based on your use case and network addressing plan.

2.2.5 - Check whether dockershim removal affects you

The `dockershim` component of Kubernetes allows the use of Docker as a Kubernetes's container runtime. Kubernetes' built-in `dockershim` component was removed in release v1.24.

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

Finding if your app has a dependencies on Docker

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

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

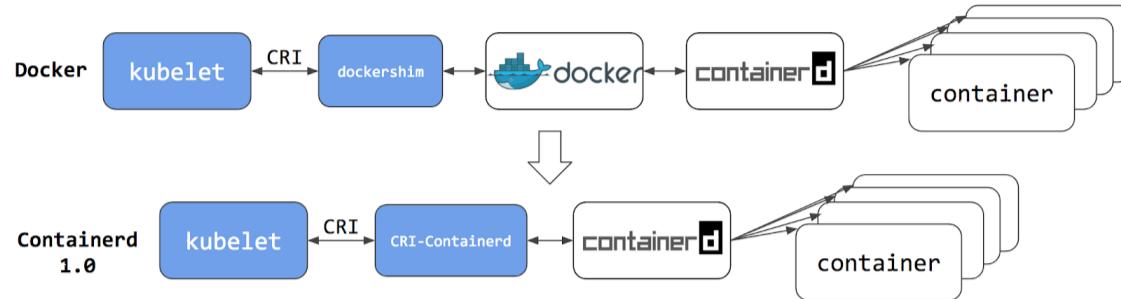
1. Make sure no privileged Pods execute Docker commands (like `docker ps`), restart the Docker service (commands such as `systemctl restart docker.service`), or modify Docker-specific files such as `/etc/docker/daemon.json`.
2. Check for any private registries or image mirror settings in the Docker configuration file (like `/etc/docker/daemon.json`). Those typically need to be reconfigured for another container runtime.
3. Check that scripts and apps running on nodes outside of your Kubernetes infrastructure do not execute Docker commands. It might be:
 - o SSH to nodes to troubleshoot;
 - o Node startup scripts;
 - o Monitoring and security agents installed on nodes directly.
4. Third-party tools that perform above mentioned privileged operations. See [Migrating telemetry and security agents from dockershim](#) for more information.
5. Make sure there are no indirect dependencies on dockershim behavior. This is an edge case and unlikely to affect your application. Some tooling may be configured to react to Docker-specific behaviors, for example, raise alert on specific metrics or search for a specific log message as part of troubleshooting instructions. If you have such tooling configured, test the behavior on a test cluster before migration.

Dependency on Docker explained

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

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

You can read about it in [Kubernetes Containerd integration goes GA](#) blog post.



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

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

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

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

Known issues

Some filesystem metrics are missing and the metrics format is different

The Kubelet `/metrics/cadvisor` endpoint provides Prometheus metrics, as documented in [Metrics for Kubernetes system components](#). If you install a metrics collector that depends on that endpoint, you might see the following issues:

- The metrics format on the Docker node is `k8s_<container-name>_<pod-name>_<namespace>_<pod-uid>_<restart-count>` but the format on other runtime is different. For example, on containerd node it is `<container-id>`.
- Some filesystem metrics are missing, as follows:

```
container_fs_inodes_free
container_fs_inodes_total
container_fs_io_current
container_fs_io_time_seconds_total
container_fs_io_time_weighted_seconds_total
container_fs_limit_bytes
container_fs_read_seconds_total
container_fs_reads_merged_total
container_fs_sector_reads_total
container_fs_sector_writes_total
container_fs_usage_bytes
container_fs_write_seconds_total
container_fs_writes_merged_total
```

Workaround

You can mitigate this issue by using [cAdvisor](#) as a standalone daemonset.

1. Find the latest [cAdvisor release](#) with the name pattern `vx.y.z-
containerd-cri` (for example, `v0.42.0-containerd-cri`).
2. Follow the steps in [cAdvisor Kubernetes Daemonset](#) to create the
daemonset.
3. Point the installed metrics collector to use the cAdvisor `/metrics`
endpoint which provides the full set of [Prometheus container
metrics](#).

Alternatives:

- Use alternative third party metrics collection solution.
- Collect metrics from the Kubelet summary API that is served at
`/stats/summary` .

What's next

- Read [Migrating from dockershim](#) to understand your next steps
- Read the [dockershim deprecation FAQ](#) article for more information.

2.2.6 - Migrating telemetry and security agents from dockershim

Note: This section links to third party projects that provide functionality required by Kubernetes. The Kubernetes project authors aren't responsible for these projects, which are listed alphabetically. To add a project to this list, read the [content guide](#) before submitting a change. [More information.](#)

Kubernetes' support for direct integration with Docker Engine is deprecated and has been removed. Most apps do not have a direct dependency on runtime hosting containers. However, there are still a lot of telemetry and monitoring agents that have a dependency on Docker to collect containers metadata, logs, and metrics. This document aggregates information on how to detect these dependencies as well as links on how to migrate these agents to use generic tools or alternative runtimes.

Telemetry and security agents

Within a Kubernetes cluster there are a few different ways to run telemetry or security agents. Some agents have a direct dependency on Docker Engine when they run as DaemonSets or directly on nodes.

Why do some telemetry agents communicate with Docker Engine?

Historically, Kubernetes was written to work specifically with Docker Engine. Kubernetes took care of networking and scheduling, relying on Docker Engine for launching and running containers (within Pods) on a node. Some information that is relevant to telemetry, such as a pod name, is only available from Kubernetes components. Other data, such as container metrics, is not the responsibility of the container runtime. Early telemetry agents needed to query the container runtime *and* Kubernetes to report an accurate picture. Over time, Kubernetes gained the ability to support multiple runtimes, and now supports any runtime that is compatible with the [container runtime interface](#).

Some telemetry agents rely specifically on Docker Engine tooling. For example, an agent might run a command such as `docker ps` or `docker top` to list containers and processes or `docker logs` to receive streamed logs. If nodes in your existing cluster use Docker Engine, and you switch to a different container runtime, these commands will not work any longer.

Identify DaemonSets that depend on Docker Engine

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

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

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

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

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

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

Detecting Docker dependency from node agents

If your cluster nodes are customized and install additional security and telemetry agents on the node, check with the agent vendor to verify whether it has any dependency on Docker.

Telemetry and security agent vendors

This section is intended to aggregate information about various telemetry and security agents that may have a dependency on container runtimes.

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

Migration from dockershim

[Aqua](#)

No changes are needed: everything should work seamlessly on the runtime switch.

[Datadog](#)

How to migrate: [Docker deprecation in Kubernetes](#) The pod that accesses Docker Engine may have a name containing any of:

- datadog-agent
- datadog
- dd-agent

[Dynatrace](#)

How to migrate: [Migrating from Docker-only to generic container metrics in Dynatrace](#)

Containerd support announcement: [Get automated full-stack visibility into containerd-based Kubernetes environments](#)

CRI-O support announcement: [Get automated full-stack visibility into your CRI-O Kubernetes containers \(Beta\)](#)

The pod accessing Docker may have name containing:

- dynatrace-oneagent

[Falco](#)

How to migrate:

[Migrate Falco from dockershim](#) Falco supports any CRI-compatible runtime (containerd is used in the default configuration); the documentation explains all details. The pod accessing Docker may have name containing:

- falco

[Prisma Cloud Compute](#)

Check [documentation for Prisma Cloud](#), under the "Install Prisma Cloud on a CRI (non-Docker) cluster" section. The pod accessing Docker may be named like:

- twistlock-defender-ds

[SignalFx \(Splunk\)](#)

The SignalFx Smart Agent (deprecated) uses several different monitors for Kubernetes including `kubernetes-cluster` , `kubelet-stats/kubelet-metrics` , and `docker-container-stats` . The `kubelet-stats` monitor was previously deprecated by the vendor, in favor of `kubelet-metrics` . The `docker-container-stats` monitor is the one affected by dockershim removal. Do not use the `docker-container-stats` with container runtimes other than Docker Engine.

How to migrate from dockershim-dependent agent:

1. Remove `docker-container-stats` from the list of [configured monitors](#).
Note, keeping this monitor enabled with non-dockershim runtime will result in incorrect metrics being reported when docker is installed on node and no metrics when docker is not installed.
2. [Enable and configure kubelet-metrics](#) monitor.

Note: The set of collected metrics will change. Review your alerting rules and dashboards.

The Pod accessing Docker may be named something like:

- signalfx-agent

[Yahoo Kubectl Flame](#)

Flame does not support container runtimes other than Docker. See <https://github.com/yahoo/kubectl-flame/issues/51>

2.3 - Generate Certificates Manually

When using client certificate authentication, you can generate certificates manually through [easyrsa](#), [openssl](#) or [cfssl](#).

easyrsa

easyrsa can manually generate certificates for your cluster.

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

```
curl -LO https://dl.k8s.io/easy-rsa/easy-rsa.tar.gz
tar xzf easy-rsa.tar.gz
cd easy-rsa-master/easyrsa3
./easyrsa init-pki
```

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

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

3. Generate server certificate and key.

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

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

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

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

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

openssl

openssl can manually generate certificates for your cluster.

1. Generate a ca.key with 2048bit:

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

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

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

3. Generate a server.key with 2048bit:

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

4. Create a config file for generating a Certificate Signing Request (CSR).

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

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

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

[ req_ext ]
subjectAltName = @alt_names

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

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

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

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

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

```
openssl x509 -req -in server.csr -CA ca.crt -CAkey ca.key \
-CAcreateserial -out server.crt -days 10000 \
-extensions v3_ext -extfile csr.conf -sha256
```

7. View the certificate signing request:

```
openssl req -noout -text -in ./server.csr
```

8. View the certificate:

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

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

cfssl

cfssl is another tool for certificate generation.

1. Download, unpack and prepare the command line tools as shown below.

Note that you may need to adapt the sample commands based on the hardware architecture and cfssl version you are using.

```
curl -L https://github.com/cloudflare/cfssl/releases/download/v1.0.0/cfssl \
chmod +x cfssl
curl -L https://github.com/cloudflare/cfssl/releases/download/v1.0.0/cfssljson \
chmod +x cfssljson
curl -L https://github.com/cloudflare/cfssl/releases/download/v1.0.0/cfssl-certinfo \
chmod +x cfssl-certinfo
```

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

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

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

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

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

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

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

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

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

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

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

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

Distributing Self-Signed CA Certificate

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

On each client, perform the following operations:

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

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

Certificates API

You can use the `certificates.k8s.io` API to provision x509 certificates to use for authentication as documented in the [Managing TLS in a cluster](#) task page.

2.4 - Manage Memory, CPU, and API Resources

2.4.1 - Configure Default Memory Requests and Limits for a Namespace

Define a default memory resource limit for a namespace, so that every new Pod in that namespace has a memory resource limit configured.

This page shows how to configure default memory requests and limits for a namespace.

A Kubernetes cluster can be divided into namespaces. Once you have a namespace that has a default memory limit, and you then try to create a Pod with a container that does not specify its own memory limit, then the control plane assigns the default memory limit to that container.

Kubernetes assigns a default memory request under certain conditions that are explained later in this topic.

Before you begin

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

- [Killercoda](#)
- [Play with Kubernetes](#)

You must have access to create namespaces in your cluster.

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 a manifest for an example LimitRange. The manifest specifies a default memory request and a default memory limit.

[admin/resource/memory-defaults.yaml](#) 

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

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

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

Now if you create a Pod in the default-mem-example namespace, and any container within that Pod does not specify its own values for memory request and memory limit, then the control plane applies default values: a memory request of 256MiB and a memory limit of 512MiB.

Here's an example manifest for a Pod that has one container. The container does not specify a memory request and limit.

[admin/resource/memory-defaults-pod.yaml](#) 

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

Create the Pod.

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

View detailed information about the Pod:

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

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 a manifest for a Pod that has one container. The container specifies a memory limit, but not a request:

[admin/resource/memory-defaults-pod-2.yaml](#) 

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

Create the Pod:

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

View detailed information about the Pod:

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

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 a manifest for a Pod that has one container. The container specifies a memory request, but not a limit:

[admin/resource/memory-defaults-pod-3.yaml](#)

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

Create the Pod:

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

View the Pod's specification:

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

The output shows that the container's memory request is set to the value specified in the container's manifest. The container is limited to use no more than 512MiB of memory, which matches the default memory limit for the namespace.

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

Note: A `LimitRange` does **not** check the consistency of the default values it applies. This means that a default value for the `limit` that is set by `LimitRange` may be less than the `request` value specified for the container in the spec that a client submits to the API server. If that

happens, the final Pod will not be scheduleable. See [Constraints on resource limits and requests](#) for more details.

Motivation for default memory limits and requests

If your namespace has a memory [resource quota](#) configured, it is helpful to have a default value in place for memory limit. Here are three of the restrictions that a resource quota imposes on a namespace:

- For every Pod that runs in the namespace, the Pod and each of its containers must have a memory limit. (If you specify a memory limit for every container in a Pod, Kubernetes can infer the Pod-level memory limit by adding up the limits for its containers).
- Memory limits apply a resource reservation on the node where the Pod in question is scheduled. The total amount of memory reserved for all Pods in the namespace must not exceed a specified limit.
- The total amount of memory actually used by all Pods in the namespace must also not exceed a specified limit.

When you add a LimitRange:

If any Pod in that namespace that includes a container does not specify its own memory limit, the control plane applies the default memory limit to that container, and the Pod can be allowed to run in a namespace that is restricted by a memory ResourceQuota.

Clean up

Delete your namespace:

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

What's next

For cluster administrators

- [Configure Default CPU Requests and Limits for a Namespace](#)
- [Configure Minimum and Maximum Memory Constraints for a Namespace](#)
- [Configure Minimum and Maximum CPU Constraints for a Namespace](#)
- [Configure Memory and CPU Quotas for a Namespace](#)
- [Configure a Pod Quota for a Namespace](#)
- [Configure Quotas for API Objects](#)

For app developers

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

2.4.2 - Configure Default CPU Requests and Limits for a Namespace

Define a default CPU resource limits for a namespace, so that every new Pod in that namespace has a CPU resource limit configured.

This page shows how to configure default CPU requests and limits for a namespace.

A Kubernetes cluster can be divided into namespaces. If you create a Pod within a namespace that has a default CPU [limit](#), and any container in that Pod does not specify its own CPU limit, then the control plane assigns the default CPU limit to that container.

Kubernetes assigns a default CPU [request](#), but only under certain conditions that are explained later in this page.

Before you begin

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

- [KillerCoda](#)
- [Play with Kubernetes](#)

You must have access to create namespaces in your cluster.

If you're not already familiar with what Kubernetes means by 1.0 CPU, read [meaning of 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 default-cpu-example
```

Create a LimitRange and a Pod

Here's a manifest for an example [LimitRange](#). The manifest specifies a default CPU request and a default CPU limit.

```
admin/resource/cpu-defaults.yaml
```

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

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

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

Now if you create a Pod in the default-cpu-example namespace, and any container in that Pod does not specify its own values for CPU request and CPU limit, then the control plane applies default values: a CPU request of 0.5 and a default CPU limit of 1.

Here's a manifest for a Pod that has one container. The container does not specify a CPU request and limit.

[admin/resource/cpu-defaults-pod.yaml](#) 

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

Create the Pod.

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

View the Pod's specification:

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

The output shows that the Pod's only container has a CPU request of 500m `cpu` (which you can read as "500 millicpu"), 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 a manifest for a Pod that has one container. The container specifies a CPU limit, but not a request:

[admin/resource/cpu-defaults-pod-2.yaml](#) 

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

Create the Pod:

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

View the [specification](#) of the Pod that you created:

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

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 an example manifest for a Pod that has one container. The container specifies a CPU request, but not a limit:

```
admin/resource/cpu-defaults-pod-3.yaml ┌─
```

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

Create the Pod:

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

View the specification of the Pod that you created:

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

The output shows that the container's CPU request is set to the value you specified at the time you created the Pod (in other words: it matches the manifest). However, the same container's CPU limit is set to 1 `cpu`, which is the default CPU limit for that namespace.

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

Motivation for default CPU limits and requests

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

- For every Pod that runs in the namespace, each of its containers must have a CPU limit.
- CPU limits apply a resource reservation on the node where the Pod in question is scheduled. The total amount of CPU that is reserved for use by all Pods in the namespace must not exceed a specified limit.

When you add a LimitRange:

If any Pod in that namespace that includes a container does not specify its own CPU limit, the control plane applies the default CPU limit to that container, and the Pod can be allowed to run in a namespace that is restricted by a CPU ResourceQuota.

Clean up

Delete your namespace:

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

What's next

For cluster administrators

- [Configure Default Memory Requests and Limits for a Namespace](#)
- [Configure Minimum and Maximum Memory Constraints for a Namespace](#)
- [Configure Minimum and Maximum CPU Constraints for a Namespace](#)
- [Configure Memory and CPU Quotas for a Namespace](#)
- [Configure a Pod Quota for a Namespace](#)
- [Configure Quotas for API Objects](#)

For app developers

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

2.4.3 - Configure Minimum and Maximum Memory Constraints for a Namespace

Define a range of valid memory resource limits for a namespace, so that every new Pod in that namespace falls within the range you configure.

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

Before you begin

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

- [KillerCoda](#)
- [Play with Kubernetes](#)

You must have access to create namespaces in your cluster.

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

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 an example manifest for a LimitRange:

[admin/resource/memory-constraints.yaml](#) 

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

Create the LimitRange:

```
kubectl apply -f https://k8s.io/examples/admin/resource/memory-constra
```

View detailed information about the LimitRange:

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

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 you define a Pod within the constraints-mem-example namespace, Kubernetes performs these steps:

- If any container in that Pod does not specify its own memory request and limit, the control plane assigns the default memory request and limit to that container.
- Verify that every container in that Pod requests at least 500 MiB of memory.
- Verify that every container in that Pod requests no more than 1024 MiB (1 GiB) of memory.

Here's a manifest for a Pod that has one container. Within the Pod spec, the sole container specifies a memory request of 600 MiB and a memory limit of 800 MiB. These satisfy the minimum and maximum memory constraints imposed by the LimitRange.

[admin/resource/memory-constraints-pod.yaml](#) 

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

Create the Pod:

```
kubectl apply -f https://k8s.io/examples/admin/resource/memory-constra
```

Verify that the Pod is running and that its container is healthy:

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

View detailed information about the Pod:

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

The output shows that the container within that Pod has a memory request of 600 MiB and a memory limit of 800 MiB. These satisfy the constraints imposed by the LimitRange for this namespace:

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

Delete your Pod:

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

Attempt to create a Pod that exceeds the maximum memory constraint

Here's a manifest for a Pod that has one container. The container specifies a memory request of 800 MiB and a memory limit of 1.5 GiB.

[admin/resource/memory-constraints-pod-2.yaml](#)

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

Attempt to create the Pod:

```
kubectl apply -f https://k8s.io/examples/admin/resource/memory-constra
```

The output shows that the Pod does not get created, because it defines a container that requests more memory than is allowed:

```
Error from server (Forbidden): error when creating "examples/admin/reso
pods "constraints-mem-demo-2" is forbidden: maximum memory usage per Co
```

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

Here's a manifest for a Pod that has one container. That container specifies a memory request of 100 MiB and a memory limit of 800 MiB.

[admin/resource/memory-constraints-pod-3.yaml](#)

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

Attempt to create the Pod:

```
kubectl apply -f https://k8s.io/examples/admin/resource/memory-constra:
```

The output shows that the Pod does not get created, because it defines a container that requests less memory than the enforced minimum:

```
Error from server (Forbidden): error when creating "examples/admin/resource/memory-constraints-mem-demo-3.yaml": pods "constraints-mem-demo-3" is forbidden: minimum memory usage per Co
```

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

Here's a manifest for a Pod that has one container. The container does not specify a memory request, and it does not specify a memory limit.

[admin/resource/memory-constraints-pod-4.yaml](#) 

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

Create the Pod:

```
kubectl apply -f https://k8s.io/examples/admin/resource/memory-constra:
```

View detailed information about the Pod:

```
kubectl get pod constraints-mem-demo-4 --namespace=constraints-mem-exa
```

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

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

Because your Pod did not define any memory request and limit for that container, the cluster applied a [default memory request and limit](#) from the LimitRange.

This means that the definition of that Pod shows those values. You can check it using `kubectl describe`:

```
# Look for the "Requests:" section of the output
kubectl describe pod constraints-mem-demo-4 --namespace=constraints-mem
```

At this point, your Pod 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
```

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 GiB of memory. You do not want to accept any Pod that requests more than 2 GiB 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 GiB of memory, but you want development workloads to be limited to 512 MiB. 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](#)

2.4.4 - Configure Minimum and Maximum CPU Constraints for a Namespace

Define a range of valid CPU resource limits for a namespace, so that every new Pod in that namespace falls within the range you configure.

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

Before you begin

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

- [KillerCoda](#)
- [Play with Kubernetes](#)

You must have access to create namespaces in your cluster.

Each node in your cluster must have at least 1.0 CPU available for Pods. See [meaning of CPU](#) to learn what Kubernetes means by “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 a manifest for an example [LimitRange](#):

[admin/resource/cpu-constraints.yaml](#) 

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

Create the LimitRange:

```
kubectl apply -f https://k8s.io/examples/admin/resource/cpu-constraint
```

View detailed information about the LimitRange:

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

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 you create a Pod in the constraints-cpu-example namespace (or some other client of the Kubernetes API creates an equivalent Pod), Kubernetes performs these steps:

- If any container in that Pod does not specify its own CPU request and limit, the control plane assigns the default CPU request and limit to that container.
- Verify that every container in that Pod specifies a CPU request that is greater than or equal to 200 millicpu.
- Verify that every container in that Pod 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 a manifest 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 for this namespace.

[admin/resource/cpu-constraints-pod.yaml](#) 

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

Create the Pod:

```
kubectl apply -f https://k8s.io/examples/admin/resource/cpu-constraint
```

Verify that the Pod is running and that its container is healthy:

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

View detailed information about the Pod:

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

The output shows that the Pod's only 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-ex
```

Attempt to create a Pod that exceeds the maximum CPU constraint

Here's a manifest for a Pod that has one container. The container specifies a CPU request of 500 millicpu and a cpu limit of 1.5 cpu.

[admin/resource/cpu-constraints-pod-2.yaml](#)

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

Attempt to create the Pod:

```
kubectl apply -f https://k8s.io/examples/admin/resource/cpu-constraint
```

The output shows that the Pod does not get created, because it defines an unacceptable container. That container is not acceptable because it specifies a CPU limit that is too large:

```
Error from server (Forbidden): error when creating "examples/admin/reso
pods "constraints-cpu-demo-2" is forbidden: maximum cpu usage per Conta
```

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

Here's a manifest for a Pod that has one container. The container specifies a CPU request of 100 millicpu and a CPU limit of 800 millicpu.

[admin/resource/cpu-constraints-pod-3.yaml](#)

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

Attempt to create the Pod:

```
kubectl apply -f https://k8s.io/examples/admin/resource/cpu-constraint
```

The output shows that the Pod does not get created, because it defines an unacceptable container. That container is not acceptable because it specifies a CPU request that is lower than the enforced minimum:

```
Error from server (Forbidden): error when creating "examples/admin/resource/pods "constraints-cpu-demo-3" is forbidden: minimum cpu usage per Container must be at least 100m
```

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

Here's a manifest for a Pod that has one container. The container does not specify a CPU request, nor does it specify a CPU limit.

[admin/resource/cpu-constraints-pod-4.yaml](#) 

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

Create the Pod:

```
kubectl apply -f https://k8s.io/examples/admin/resource/cpu-constraint
```

View detailed information about the Pod:

```
kubectl get pod constraints-cpu-demo-4 --namespace=constraints-cpu-exam
```

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

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

Because that container did not specify its own CPU request and limit, the control plane applied the [default CPU request and limit](#) from the LimitRange for this namespace.

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

Delete your Pod:

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

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](#)

2.4.5 - Configure Memory and CPU Quotas for a Namespace

Define overall memory and CPU resource limits for a namespace.

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

Before you begin

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

- [Killercoda](#)
- [Play with Kubernetes](#)

You must have access to create namespaces in your cluster.

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 a manifest for an example ResourceQuota:

[admin/resource/quota-mem-cpu.yaml](#) 

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

Create the ResourceQuota:

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

View detailed information about the ResourceQuota:

```
kubectl get resourcequota mem-cpu-demo --namespace=quota-mem-cpu-examp
```

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

- For every Pod in the namespace, each container must have a memory request, memory limit, cpu request, and cpu limit.
- The memory request total for all Pods in that namespace must not exceed 1 GiB.
- The memory limit total for all Pods in that namespace must not exceed 2 GiB.
- The CPU request total for all Pods in that namespace must not exceed 1 cpu.
- The CPU limit total for all Pods in that namespace must not exceed 2 cpu.

See [meaning of CPU](#) to learn what Kubernetes means by "1 CPU".

Create a Pod

Here is a manifest for an example Pod:

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



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

Create the Pod:

```
kubectl apply -f https://k8s.io/examples/admin/resource/quota-mem-cpu-1
```

Verify that the Pod is running and that its (only) container is healthy:

```
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-examp
```

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
```

If you have the `jq` tool, you can also query (using [JSONPath](#)) for just the `used` values, **and** pretty-print that of the output. For example:

```
kubectl get resourcequota mem-cpu-demo --namespace=quota-mem-cpu-examp
```

Attempt to create a second Pod

Here is a manifest for a second Pod:

[admin/resource/quota-mem-cpu-pod-2.yaml](#) 

```
apiVersion: v1  
kind: Pod  
metadata:  
  name: quota-mem-cpu-demo-2  
spec:  
  containers:  
  - name: quota-mem-cpu-demo-2-ctr  
    image: redis  
    resources:  
      limits:  
        memory: "1Gi"  
        cpu: "800m"  
      requests:  
        memory: "700Mi"  
        cpu: "400m"
```

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

1 GiB.

Attempt to create the Pod:

```
kubectl apply -f https://k8s.io/examples/admin/resource/quota-mem-cpu-1
```

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

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

Discussion

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

Instead of managing total resource use within a namespace, you might want to restrict individual Pods, or the containers in those Pods. To achieve that kind of limiting, 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](#)

2.4.6 - Configure a Pod Quota for a Namespace

Restrict how many Pods you can create within a namespace.

This page shows how to set a quota for the total number of Pods that can run in a Namespace. You specify quotas in a [ResourceQuota](#) object.

Before you begin

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

- [Killercoda](#)
- [Play with Kubernetes](#)

You must have access to create namespaces in your cluster.

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 an example manifest for a ResourceQuota:

```
admin/resource/quota-pod.yaml 
```

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

Create the ResourceQuota:

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

View detailed information about the ResourceQuota:

```
kubectl get resourcequota pod-demo --namespace=quota-pod-example --out
```

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 an example manifest for a Deployment:

[admin/resource/quota-pod-deployment.yaml](#)

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

In that manifest, `replicas: 3` tells Kubernetes to attempt to create three new Pods, all running the same application.

Create the Deployment:

```
kubectl apply -f https://k8s.io/examples/admin/resource/quota-pod-depl
```

View detailed information about the Deployment:

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

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

```
spec:  
...  
replicas: 3  
...  
status:  
availableReplicas: 2  
...  
lastUpdateTime: 2021-04-02T20:57:05Z  
message: 'unable to create pods: pods "pod-quota-demo-1650323038-"  
exceeded quota: pod-demo, requested: pods=1, used: pods=2, limit:  
...  
...'
```

Choice of resource

In this task you have defined a ResourceQuota that limited the total number of Pods, but you could also limit the total number of other kinds of object. For example, you might decide to limit how many CronJobs that can live in a single namespace.

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](#)

2.5 - Install a Network Policy Provider

2.5.1 - Use Antrea for NetworkPolicy

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

Before you begin

You need to have a Kubernetes cluster. Follow the [kubeadm getting started guide](#) to bootstrap one.

Deploying Antrea with kubeadm

Follow [Getting Started](#) guide to deploy Antrea for kubeadm.

What's next

Once your cluster is running, you can follow the [Declare Network Policy](#) to try out Kubernetes NetworkPolicy.

2.5.2 - Use Calico for NetworkPolicy

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

Before you begin

Decide whether you want to deploy a [cloud](#) or [local](#) cluster.

Creating a Calico cluster with Google Kubernetes Engine (GKE)

Prerequisite: [gcloud](#).

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

Syntax

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

Example

```
gcloud container clusters create my-calico-cluster --enable-netwo
```

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.

2.5.3 - Use Cilium for NetworkPolicy

This page shows how to use Cilium for NetworkPolicy.

For background on Cilium, read the [Introduction to Cilium](#).

Before you begin

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

- [Killercoda](#)
- [Play with Kubernetes](#)

To check the version, enter `kubectl version`.

Deploying Cilium on Minikube for Basic Testing

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

To start minikube, minimal version required is $\geq v1.5.2$, run the with the following arguments:

```
minikube version
```

```
minikube version: v1.5.2
```

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

For minikube you can install Cilium using its CLI tool. To do so, first download the latest version of the CLI with the following command:

```
curl -LO https://github.com/cilium/cilium-cli/releases/latest/download,
```

Then extract the downloaded file to your `/usr/local/bin` directory with the following command:

```
sudo tar xzvfC cilium-linux-amd64.tar.gz /usr/local/bin  
rm cilium-linux-amd64.tar.gz
```

After running the above commands, you can now install Cilium with the following command:

```
cilium install
```

Cilium will then automatically detect the cluster configuration and create and install the appropriate components for a successful installation. The components are:

- Certificate Authority (CA) in Secret `cilium-ca` and certificates for Hubble (Cilium's observability layer).
- Service accounts.
- Cluster roles.
- ConfigMap.
- Agent DaemonSet and an Operator Deployment.

After the installation, you can view the overall status of the Cilium deployment with the `cilium status` command. See the expected output of the `status` command [here](#).

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 -l k8s-app=cilium
```

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

NAME	READY	STATUS	RESTARTS	AGE
cilium-kkdhz	1/1	Running	0	3m23s
...				

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

What's next

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

2.5.4 - Use Kube-router for NetworkPolicy

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

Before you begin

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

Installing Kube-router addon

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

What's next

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

2.5.5 - Romana for NetworkPolicy

This page shows how to use Romana for NetworkPolicy.

Before you begin

Complete steps 1, 2, and 3 of the [kubeadm getting started guide](#).

Installing Romana with kubeadm

Follow the [containerized installation guide](#) for kubeadm.

Applying network policies

To apply network policies use one of the following:

- [Romana network policies](#).
 - [Example of Romana network policy](#).
- The NetworkPolicy API.

What's next

Once you have installed Romana, you can follow the [Declare Network Policy](#) to try out Kubernetes NetworkPolicy.

2.5.6 - Weave Net for NetworkPolicy

This page shows how to use Weave Net for NetworkPolicy.

Before you begin

You need to have a Kubernetes cluster. Follow the [kubeadm getting started guide](#) to bootstrap one.

Install the Weave Net addon

Follow the [Integrating Kubernetes via the Addon](#) guide.

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

Test the installation

Verify that the weave works.

Enter the following command:

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

The output is similar to this:

NAME	READY	STATUS	RESTARTS
weave-net-1t1qg	2/2	Running	0
weave-net-231d7	2/2	Running	1
weave-net-7nmwrt	2/2	Running	3
weave-net-pmw8w	2/2	Running	0

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](#).

2.6 - Access Clusters Using the Kubernetes API

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

Before you begin

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

- [Killercoda](#)
- [Play with Kubernetes](#)

To check the version, enter `kubectl version`.

Accessing the Kubernetes API

Accessing for the first time with kubectl

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

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

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

```
kubectl config view
```

Many of the [examples](#) provide an introduction to using `kubectl`. Complete documentation is found in the [kubectl manual](#).

Directly accessing the REST API

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

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

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

Using kubectl proxy

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

Run it like this:

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

See [kubectl proxy](#) for more details.

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

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

The output is similar to this:

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

Without kubectl proxy

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

Using grep/cut approach:

```
# Check all possible clusters, as your .KUBECONFIG may have multiple configurations
kubectl config view -o jsonpath='{["Cluster name\tServer\n"]}{range .clusters[*]}{.name} {.server}\n{end}' > /dev/null

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

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

# Create a secret to hold a token for the default service account
kubectl apply -f - <<EOF
apiVersion: v1
kind: Secret
metadata:
  name: default-token
  annotations:
    kubernetes.io/service-account.name: default
type: kubernetes.io/service-account-token
EOF

# Wait for the token controller to populate the secret with a token:
while ! kubectl describe secret default-token | grep -E '^token' >/dev/null; do
  echo "waiting for token..." >&2
  sleep 1
done

# Get the token value
TOKEN=$(kubectl get secret default-token -o jsonpath='{.data.token}' | base64 --decode)
```

The output is similar to this:

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

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

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

Programmatic access to the API

Kubernetes officially supports client libraries for [Go](#), [Python](#), [Java](#), [dotnet](#), [JavaScript](#), and [Haskell](#). There are other client libraries that are provided and maintained by their authors, not the Kubernetes team. See [client](#)

[libraries](#) for accessing the API from other languages and how they authenticate.

Go client

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

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

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

```
package main

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

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

If the application is deployed as a Pod in the cluster, see [Accessing the API from within a Pod](#).

Python client

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

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

```
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.met
```

Java client

To install the [Java Client](#), run:

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

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

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

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

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

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

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

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

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

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

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

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

dotnet client

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

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

```
using System;
using k8s;

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

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

JavaScript client

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

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

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

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

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

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

Haskell client

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

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

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

What's next

- [Accessing the Kubernetes API from a Pod](#)

2.7 - Advertise Extended Resources for a Node

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

Before you begin

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

- [Killercoda](#)
- [Play with Kubernetes](#)

To check the version, enter `kubectl version`.

Get the names of your Nodes

```
kubectl get nodes
```

Choose one of your Nodes to use for this exercise.

Advertise a new extended resource on one of your Nodes

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

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

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

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

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

```
kubectl proxy
```

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

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

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

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

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

Describe your Node:

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

Once again, the output shows the dongle resource:

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

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

Discussion

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

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

Storage example

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

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

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

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

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

Clean up

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

```
PATCH /api/v1/nodes/<your-node-name>/status HTTP/1.1  
Accept: application/json  
Content-Type: application/json-patch+json  
Host: k8s-master:8080  
  
[  
 {  
   "op": "remove",  
   "path": "/status/capacity/example.com~1dongle",  
 }  
]
```

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

```
kubectl proxy
```

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

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

Verify that the dongle advertisement has been removed:

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

(you should not see any output)

What's next

For application developers

- [Assign Extended Resources to a Container](#)

For cluster administrators

- [Configure Minimum and Maximum Memory Constraints for a Namespace](#)
- [Configure Minimum and Maximum CPU Constraints for a Namespace](#)

2.8 - Autoscale the DNS Service in a Cluster

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

Before you begin

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

- [Killercoda](#)
- [Play with Kubernetes](#)

To check the version, enter `kubectl version`.

- This guide assumes your nodes use the AMD64 or Intel 64 CPU architecture.
- Make sure [Kubernetes DNS](#) is enabled.

Determine whether DNS horizontal autoscaling is already enabled

List the [Deployments](#) in your cluster in the `kube-system` namespace:

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

The output is similar to this:

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

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

Get the name of your DNS Deployment

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

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

The output is similar to this:

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

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

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

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

Your scale target is

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

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

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

Enable DNS horizontal autoscaling

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

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

[admin/dns/dns-horizontal-autoscaler.yaml](#) 

```
kind: ServiceAccount
apiVersion: v1
metadata:
  name: kube-dns-autoscaler
  namespace: kube-system
---
kind: ClusterRole
apiVersion: rbac.authorization.k8s.io/v1
metadata:
  name: system:kube-dns-autoscaler
rules:
- apiGroups: [""]
  resources: ["nodes"]
  verbs: ["list", "watch"]
- apiGroups: [""]
  resources: ["replicationcontrollers/scale"]
  verbs: ["get", "update"]
- apiGroups: ["apps"]
  resources: ["deployments/scale", "replicasets/scale"]
  verbs: ["get", "update"]
# Remove the configmaps rule once below issue is fixed:
# kubernetes-incubator/cluster-proportional-autoscaler#16
- apiGroups: [""]
  resources: ["configmaps"]
  verbs: ["get", "create"]
---
kind: ClusterRoleBinding
apiVersion: rbac.authorization.k8s.io/v1
metadata:
  name: system:kube-dns-autoscaler
subjects:
- kind: ServiceAccount
  name: kube-dns-autoscaler
  namespace: kube-system
roleRef:
  kind: ClusterRole
  name: system:kube-dns-autoscaler
  apiGroup: rbac.authorization.k8s.io
---
apiVersion: apps/v1
kind: Deployment
metadata:
  name: kube-dns-autoscaler
  namespace: kube-system
  labels:
    k8s-app: kube-dns-autoscaler
    kubernetes.io/cluster-service: "true"
spec:
  selector:
    matchLabels:
      k8s-app: kube-dns-autoscaler
  template:
    metadata:
      labels:
        k8s-app: kube-dns-autoscaler
    spec:
      priorityClassName: system-cluster-critical
      securityContext:
        seccompProfile:
          type: RuntimeDefault
      supplementalGroups: [ 65534 ]
      fsGroup: 65534
      nodeSelector:
        kubernetes.io/os: linux
      containers:
```

```

- name: autoscaler
  image: registry.k8s.io/cpa/cluster-proportional-autoscaler:1.8
  resources:
    requests:
      cpu: "20m"
      memory: "10Mi"
  command:
    - /cluster-proportional-autoscaler
    - --namespace=kube-system
    - --configmap=kube-dns-autoscaler
    # Should keep target in sync with cluster/addons/dns/kube-dn
    - --target=<SCALE_TARGET>
    # When cluster is using Large nodes(with more cores), "cores
    # If using small nodes, "nodesPerReplica" should dominate.
    - --default-params={"linear":{"coresPerReplica":256,"nodesPe
    - --logtostderr=true
    - --v=2
  tolerations:
    - key: "CriticalAddonsOnly"
      operator: "Exists"
  serviceAccountName: kube-dns-autoscaler

```

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

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

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

The output of a successful command is:

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

DNS horizontal autoscaling is now enabled.

Tune DNS autoscaling parameters

Verify that the dns-autoscaler ConfigMap exists:

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

The output is similar to this:

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

Modify the data in the ConfigMap:

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

Look for this line:

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

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

```
replicas = max( ceil( cores * 1/coresPerReplica ) , ceil( nodes * 1/nod
```

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

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

There are other supported scaling patterns. For details, see [cluster-proportional-autoscaler](#).

Disable DNS horizontal autoscaling

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

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

This option works for all situations. Enter this command:

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

The output is:

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

Verify that the replica count is zero:

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

The output displays 0 in the DESIRED and CURRENT columns:

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

Option 2: Delete the dns-autoscaler deployment

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

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

The output is:

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

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

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

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

```
/etc/kubernetes/addons/dns-horizontal-autoscaler/dns-horizontal-autosca
```

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

Understanding how DNS horizontal autoscaling works

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

What's next

- Read about [Guaranteed Scheduling For Critical Add-On Pods](#).
- Learn more about the [implementation of cluster-proportional-autoscaler](#).

2.9 - Change the default StorageClass

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

Before you begin

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

- [Killercoda](#)
- [Play with Kubernetes](#)

To check the version, enter `kubectl version`.

Why change the default storage class?

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

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

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

Changing the default StorageClass

1. List the StorageClasses in your cluster:

```
kubectl get storageclass
```

The output is similar to this:

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

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

2. Mark the default StorageClass as non-default:

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

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

```
kubectl patch storageclass standard -p '{"metadata": {"annotation
```

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

3. Mark a StorageClass as default:

Similar to the previous step, you need to add/set the annotation

```
storageclass.kubernetes.io/is-default-class=true .
```

```
kubectl patch storageclass gold -p '{"metadata": {"annotations":{
```

Please note that at most one StorageClass can be marked as default.

If two or more of them are marked as default, a

`PersistentVolumeClaim` without `storageClassName` explicitly specified cannot be created.

4. Verify that your chosen StorageClass is default:

```
kubectl get storageclass
```

The output is similar to this:

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

What's next

- Learn more about [PersistentVolumes](#).

2.10 - Switching from Polling to CRI Event-based Updates to Container Status

FEATURE STATE: [Kubernetes v1.27 \[beta\]](#)

This page shows how to migrate nodes to use event based updates for container status. The event-based implementation reduces node resource consumption by the kubelet, compared to the legacy approach that relies on polling. You may know this feature as *evented Pod lifecycle event generator (PLEG)*. That's the name used internally within the Kubernetes project for a key implementation detail.

The polling based approach is referred to as *generic PLEG*.

Before you begin

- You need to run a version of Kubernetes that provides this feature. Kubernetes v1.27 includes beta support for event-based container status updates. The feature is beta but is *disabled* by default because it requires support from the container runtime.
- Your Kubernetes server must be at or later than version 1.26. To check the version, enter `kubectl version`. If you are running a different version of Kubernetes, check the documentation for that release.
- The container runtime in use must support container lifecycle events. The kubelet automatically switches back to the legacy generic PLEG mechanism if the container runtime does not announce support for container lifecycle events, even if you have this feature gate enabled.

Why switch to Evented PLEG?

- The *Generic PLEG* incurs non-negligible overhead due to frequent polling of container statuses.
- This overhead is exacerbated by Kubelet's parallelized polling of container states, thus limiting its scalability and causing poor performance and reliability problems.
- The goal of *Evented PLEG* is to reduce unnecessary work during inactivity by replacing periodic polling.

Switching to Evented PLEG

1. Start the Kubelet with the [feature gate](#) `EventedPLEG` enabled. You can manage the kubelet feature gates editing the kubelet [config file](#) and restarting the kubelet service. You need to do this on each node where you are using this feature.
2. Make sure the node is [drained](#) before proceeding.
3. Start the container runtime with the container event generation enabled.

[Containerd](#)

[CRI-O](#)

Version 1.7+

Your Kubernetes server must be at or later than version 1.26. To check the version, enter `kubectl version`.

4. Verify that the kubelet is using event-based container stage change monitoring. To check, look for the term `EventedPLEG` in the kubelet logs.

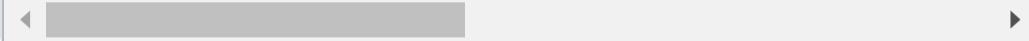
The output should be similar to this:

```
I0314 11:10:13.909915 1105457 feature_gate.go:249] feature gates:
```



If you have set `--v` to 4 and above, you might see more entries that indicate that the kubelet is using event-based container state monitoring.

```
I0314 11:12:42.009542 1110177 evented.go:238] "Evented PLEG: Gene  
I0314 11:12:44.623326 1110177 evented.go:238] "Evented PLEG: Gene  
I0314 11:12:44.714564 1110177 evented.go:238] "Evented PLEG: Gene
```



What's next

- Learn more about the design in the Kubernetes Enhancement Proposal (KEP): [Kubelet Evented PLEG for Better Performance](#).

2.11 - Change the Reclaim Policy of a PersistentVolume

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

Before you begin

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

- [Killercoda](#)
- [Play with Kubernetes](#)

To check the version, enter `kubectl version`.

Why change reclaim policy of a PersistentVolume

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

Changing the reclaim policy of a PersistentVolume

1. List the PersistentVolumes in your cluster:

```
kubectl get pv
```

The output is similar to this:

NAME	CAPACITY	ACCESSMODES
pvc-b6efd8da-b7b5-11e6-9d58-0ed433a7dd94	4Gi	RWO
pvc-b95650f8-b7b5-11e6-9d58-0ed433a7dd94	4Gi	RWO
pvc-bb3ca71d-b7b5-11e6-9d58-0ed433a7dd94	4Gi	RWO

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

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

```
kubectl patch pv <your-pv-name> -p '{"spec":{"persistentVolumeReclai
```

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

Note:

On Windows, you must *double* quote any JSONPath template that contains spaces (not single quote as shown above for bash). This in turn means that you must use a single quote or escaped double quote around any literals in the template. For example:

```
kubectl patch pv <your-pv-name> -p "{\"spec\":{\"persistentVolumeReclai
```

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

```
kubectl get pv
```

The output is similar to this:

NAME	CAPACITY	ACCESSMODES
pvc-b6efd8da-b7b5-11e6-9d58-0ed433a7dd94	4Gi	RWO
pvc-b95650f8-b7b5-11e6-9d58-0ed433a7dd94	4Gi	RWO
pvc-bb3ca71d-b7b5-11e6-9d58-0ed433a7dd94	4Gi	RWO

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](#).

References

- [PersistentVolume](#)
 - Pay attention to the `.spec.persistentVolumeReclaimPolicy` [field](#) of PersistentVolume.
- [PersistentVolumeClaim](#)

2.12 - Cloud Controller Manager Administration

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

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

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

Administration

Requirements

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

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

Running `cloud-controller-manager`

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

- `kubelet`, `kube-apiserver`, and `kube-controller-manager` must be set according to the user's usage of external CCM. If the user has an external CCM (not the internal cloud controller loops in the Kubernetes Controller Manager), then `--cloud-provider=external` must be specified. Otherwise, it should not be specified.

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

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

controller manager will hit rate limits since it is now responsible for almost all API calls to your cloud from within the cluster.

The cloud controller manager can implement:

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

Examples

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

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

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

[admin/cloud/ccm-example.yaml](#) 

```
# This is an example of how to set up cloud-controller-manager as a DaemonSet.
# It assumes that your masters can run pods and has the role node-role.kubernetes.io/leader
# Note that this Daemonset will not work straight out of the box for you.
# Meant to be a guideline.

---
apiVersion: v1
kind: ServiceAccount
metadata:
  name: cloud-controller-manager
  namespace: kube-system
---
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRoleBinding
metadata:
  name: system:cloud-controller-manager
roleRef:
  apiGroup: rbac.authorization.k8s.io
  kind: ClusterRole
  name: cluster-admin
subjects:
- kind: ServiceAccount
  name: cloud-controller-manager
  namespace: kube-system
---
apiVersion: apps/v1
kind: DaemonSet
metadata:
  labels:
    k8s-app: cloud-controller-manager
  name: cloud-controller-manager
  namespace: kube-system
spec:
  selector:
    matchLabels:
      k8s-app: cloud-controller-manager
  template:
    metadata:
      labels:
        k8s-app: cloud-controller-manager
    spec:
      serviceAccountName: cloud-controller-manager
      containers:
        - name: cloud-controller-manager
          # for in-tree providers we use registry.k8s.io/cloud-controller-manager:v1.8.0
          # this can be replaced with any other image for out-of-tree providers
          image: registry.k8s.io/cloud-controller-manager:v1.8.0
          command:
            - /usr/local/bin/cloud-controller-manager
            - --cloud-provider=[YOUR_CLOUD_PROVIDER] # Add your own cloud provider here
            - --leader-elect=true
            - --use-service-account-credentials
            # these flags will vary for every cloud provider
            - --allocate-node-cidrs=true
            - --configure-cloud-routes=true
            - --cluster-cidr=172.17.0.0/16
      tolerations:
        # this is required so CCM can bootstrap itself
        - key: node.cloudprovider.kubernetes.io/uninitialized
          value: "true"
          effect: NoSchedule
        # these tolerations are to have the daemonset runnable on control plane nodes
        # remove them if your control plane nodes should not run pods
        - key: node-role.kubernetes.io/control-plane
          operator: Exists
          effect: NoSchedule
```

```
- key: node-role.kubernetes.io/master
  operator: Exists
  effect: NoSchedule
# this is to restrict CCM to only run on master nodes
# the node selector may vary depending on your cluster setup
nodeSelector:
  node-role.kubernetes.io/master: ""
```

Limitations

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

Support for Volumes

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

Scalability

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

Chicken and Egg

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

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

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

What's next

To build and develop your own cloud controller manager, read [Developing Cloud Controller Manager](#).

2.13 - Configure a kubelet image credential provider

FEATURE STATE: [Kubernetes v1.26 \[stable\]](#)

Starting from Kubernetes v1.20, the kubelet can dynamically retrieve credentials for a container image registry using exec plugins. The kubelet and the exec plugin communicate through stdio (stdin, stdout, and stderr) using Kubernetes versioned APIs. These plugins allow the kubelet to request credentials for a container registry dynamically as opposed to storing static credentials on disk. For example, the plugin may talk to a local metadata server to retrieve short-lived credentials for an image that is being pulled by the kubelet.

You may be interested in using this capability if any of the below are true:

- API calls to a cloud provider service are required to retrieve authentication information for a registry.
- Credentials have short expiration times and requesting new credentials frequently is required.
- Storing registry credentials on disk or in imagePullSecrets is not acceptable.

This guide demonstrates how to configure the kubelet's image credential provider plugin mechanism.

Before you begin

- You need a Kubernetes cluster with nodes that support kubelet credential provider plugins. This support is available in Kubernetes 1.28; Kubernetes v1.24 and v1.25 included this as a beta feature, enabled by default.
- A working implementation of a credential provider exec plugin. You can build your own plugin or use one provided by cloud providers.

Your Kubernetes server must be at or later than version v1.26. To check the version, enter [kubectl version](#).

Installing Plugins on Nodes

A credential provider plugin is an executable binary that will be run by the kubelet. Ensure that the plugin binary exists on every node in your cluster and stored in a known directory. The directory will be required later when configuring kubelet flags.

Configuring the Kubelet

In order to use this feature, the kubelet expects two flags to be set:

- `--image-credential-provider-config` - the path to the credential provider plugin config file.
- `--image-credential-provider-bin-dir` - the path to the directory where credential provider plugin binaries are located.

Configure a kubelet credential provider

The configuration file passed into `--image-credential-provider-config` is read by the kubelet to determine which exec plugins should be invoked for which container images. Here's an example configuration file you may end

up using if you are using the [ECR-based plugin](#):

```
apiVersion: kubelet.config.k8s.io/v1
kind: CredentialProviderConfig
# providers is a list of credential provider helper plugins that will be
# Multiple providers may match against a single image, in which case credentials
# from all providers will be returned to the kubelet. If multiple providers
# for a single image, the results are combined. If providers return overlapping
# auth keys, the value from the provider earlier in this list is used.
providers:
  # name is the required name of the credential provider. It must match the
  # provider executable as seen by the kubelet. The executable must be located in
  # bin directory (set by the --image-credential-provider-bin-dir flag).
  - name: ecr-credential-provider
    # matchImages is a required list of strings used to match against the URL
    # determine if this provider should be invoked. If one of the strings matches
    # requested image from the kubelet, the plugin will be invoked and asked
    # to provide credentials. Images are expected to contain the registry
    # and URL path.
    #
    # Each entry in matchImages is a pattern which can optionally contain
    # Globs can be used in the domain, but not in the port or the path
    # as subdomains like '*.k8s.io' or 'k8s.*.io', and top-level-domains
    # Matching partial subdomains like 'app*.k8s.io' is also supported
    # a single subdomain segment, so `*.io` does **not** match `*.k8s.io`
    #
    # A match exists between an image and a matchImage when all of the
    # - Both contain the same number of domain parts and each part matches
    # - The URL path of an matchImages must be a prefix of the target URL
    # - If the matchImages contains a port, then the port must match in the target URL
    #
    # Example values of matchImages:
    # - 123456789.dkr.ecr.us-east-1.amazonaws.com
    # - *.azurecr.io
    # - gcr.io
    # - *.*.registry.io
    # - registry.io:8080/path
  matchImages:
    - "*.*.dkr.ecr.*.amazonaws.com"
    - "*.*.dkr.ecr.*.amazonaws.com.cn"
    - "*.*.dkr.ecr-fips.*.amazonaws.com"
    - "*.*.dkr.ecr.us-iso-east-1.c2s.ic.gov"
    - "*.*.dkr.ecr.us-isob-east-1.sc2s.sgov.gov"
  # defaultCacheDuration is the default duration the plugin will cache
  # if a cache duration is not provided in the plugin response. This is
  defaultCacheDuration: "12h"
  # Required input version of the exec CredentialProviderRequest. The
  # MUST use the same encoding version as the input. Current supported
  # - credentialprovider.kubelet.k8s.io/v1
apiVersion: credentialprovider.kubelet.k8s.io/v1
# Arguments to pass to the command when executing it.
# +optional
# args:
#   - --example-argument
# Env defines additional environment variables to expose to the provider
# are unioned with the host's environment, as well as variables defined
# to pass argument to the plugin.
# +optional
env:
  - name: AWS_PROFILE
    value: example_profile
```

The `providers` field is a list of enabled plugins used by the kubelet. Each entry has a few required fields:

- `name` : the name of the plugin which MUST match the name of the executable binary that exists in the directory passed into `--image-credential-provider-bin-dir` .
- `matchImages` : a list of strings used to match against images in order to determine if this provider should be invoked. More on this below.
- `defaultCacheDuration` : the default duration the kubelet will cache credentials in-memory if a cache duration was not specified by the plugin.
- `apiVersion` : the API version that the kubelet and the exec plugin will use when communicating.

Each credential provider can also be given optional args and environment variables as well. Consult the plugin implementors to determine what set of arguments and environment variables are required for a given plugin.

Configure image matching

The `matchImages` field for each credential provider is used by the kubelet to determine whether a plugin should be invoked for a given image that a Pod is using. Each entry in `matchImages` is an image pattern which can optionally contain a port and a path. Globs can be used in the domain, but not in the port or the path. Globs are supported as subdomains like `*.k8s.io` OR `k8s.*.io`, and top-level domains such as `k8s.*`. Matching partial subdomains like `app*.k8s.io` is also supported. Each glob can only match a single subdomain segment, so `*.io` does NOT match `*.k8s.io`.

A match exists between an image name and a `matchImage` entry when all of the below are true:

- Both contain the same number of domain parts and each part matches.
- The URL path of match image must be a prefix of the target image URL path.
- If the `matchImages` contains a port, then the port must match in the image as well.

Some example values of `matchImages` patterns are:

- `123456789.dkr.ecr.us-east-1.amazonaws.com`
- `*.azurecr.io`
- `gcr.io`
- `*.*.registry.io`
- `foo.registry.io:8080/path`

What's next

- Read the details about `CredentialProviderConfig` in the [kubelet configuration API \(v1\) reference](#).
- Read the [kubelet credential provider API reference \(v1\)](#).

2.14 - Configure Quotas for API Objects

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

Before you begin

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

- [KillerCoda](#)
- [Play with Kubernetes](#)

To check the version, enter `kubectl version`.

Create a namespace

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

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

Create a ResourceQuota

Here is the configuration file for a ResourceQuota object:

```
admin/resource/quota-objects.yaml 
```

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

Create the ResourceQuota:

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

View detailed information about the ResourceQuota:

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

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

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

Create a PersistentVolumeClaim

Here is the configuration file for a PersistentVolumeClaim object:

[admin/resource/quota-objects-pvc.yaml](#)

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

Create the PersistentVolumeClaim:

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

Verify that the PersistentVolumeClaim was created:

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

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

NAME	STATUS
pvc-quota-demo	Pending

Attempt to create a second PersistentVolumeClaim

Here is the configuration file for a second PersistentVolumeClaim:

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

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

Attempt to create the second PersistentVolumeClaim:

```
kubectl apply -f https://k8s.io/examples/admin/resource/quota-objects-1
```

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

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

Notes

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

String	API Object
"pods"	Pod
"services"	Service
"replicationcontrollers"	ReplicationController
"resourcequotas"	ResourceQuota
"secrets"	Secret
"configmaps"	ConfigMap
"persistentvolumeclaims"	PersistentVolumeClaim

"services.nodeports"	Service of type NodePort
"services.loadbalancers"	Service of type LoadBalancer

Clean up

Delete your namespace:

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

What's next

For cluster administrators

- [Configure Default Memory Requests and Limits for a Namespace](#)
- [Configure Default CPU Requests and Limits for a Namespace](#)
- [Configure Minimum and Maximum Memory Constraints for a Namespace](#)
- [Configure Minimum and Maximum CPU Constraints for a Namespace](#)
- [Configure Memory and CPU Quotas for a Namespace](#)
- [Configure a Pod Quota for a Namespace](#)

For app developers

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

2.15 - Control CPU Management Policies on the Node

FEATURE STATE: [Kubernetes v1.26 \[stable\]](#)

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.

For detailed information on resource management, please refer to the [Resource Management for Pods and Containers](#) documentation.

Before you begin

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

- [Killercoda](#)
- [Play with Kubernetes](#)

Your Kubernetes server must be at or later than version v1.26. To check the version, enter [kubectl version](#).

If you are running an older version of Kubernetes, please look at the documentation for the version you are actually running.

CPU Management Policies

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

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

Configuration

The CPU Manager policy is set with the `--cpu-manager-policy` kubelet flag or the `cpuManagerPolicy` field in [KubeletConfiguration](#). There are two supported policies:

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

The CPU manager periodically writes resource updates through the CRI in order to reconcile in-memory CPU assignments with cgroupfs. The reconcile frequency is set through a new Kubelet configuration value `--`

`cpu-manager-reconcile-period`. If not specified, it defaults to the same duration as `--node-status-update-frequency`.

The behavior of the static policy can be fine-tuned using the `--cpu-manager-policy-options` flag. The flag takes a comma-separated list of `key=value` policy options. If you disable the `CPUManagerPolicyOptions` [feature gate](#) then you cannot fine-tune CPU manager policies. In that case, the CPU manager operates only using its default settings.

In addition to the top-level `CPUManagerPolicyOptions` feature gate, the policy options are split into two groups: alpha quality (hidden by default) and beta quality (visible by default). The groups are guarded respectively by the `CPUManagerPolicyAlphaOptions` and `CPUManagerPolicyBetaOptions` feature gates. Diverging from the Kubernetes standard, these feature gates guard groups of options, because it would have been too cumbersome to add a feature gate for each individual option.

Changing the CPU Manager Policy

Since the CPU manager policy can only be applied when kubelet spawns new pods, simply changing from "none" to "static" won't apply to existing pods. So in order to properly change the CPU manager policy on a node, perform the following steps:

1. [Drain](#) the node.
2. Stop kubelet.
3. Remove the old CPU manager state file. The path to this file is `/var/lib/kubelet/cpu_manager_state` by default. This clears the state maintained by the CPUManager so that the cpu-sets set up by the new policy won't conflict with it.
4. Edit the kubelet configuration to change the CPU manager policy to the desired value.
5. Start kubelet.

Repeat this process for every node that needs its CPU manager policy changed. Skipping this process will result in kubelet crashlooping with the following error:

```
could not restore state from checkpoint: configured policy "static" did
```

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](#) and [Burstable pods](#) are enforced using CFS quota.

Static policy

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

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

Note: CPU Manager doesn't support offline and online of CPUs at runtime. Also, if the set of online CPUs changes on the node, the node

must be drained and CPU manager manually reset by deleting the state file `cpu_manager_state` in the kubelet root directory.

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

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

As `Guaranteed` pods whose containers fit the requirements for being statically assigned are scheduled to the node, CPUs are removed from the shared pool and placed in the cpuset for the container. CFS quota is not used to bound the CPU usage of these containers as their usage is bound by the scheduling domain itself. In 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 `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.

Static policy options

You can toggle groups of options on and off based upon their maturity level using the following feature gates:

- `CPUManagerPolicyBetaOptions` default enabled. Disable to hide beta-level options.
- `CPUManagerPolicyAlphaOptions` default disabled. Enable to show alpha-level options. You will still have to enable each option using the `CPUManagerPolicyOptions` kubelet option.

The following policy options exist for the static `CPUManager` policy:

- `full-pcpus-only` (beta, visible by default) (1.22 or higher)
- `distribute-cpus-across-numa` (alpha, hidden by default) (1.23 or higher)
- `align-by-socket` (alpha, hidden by default) (1.25 or higher)

If the `full-pcpus-only` policy option is specified, the static policy will always allocate full physical cores. By default, without this option, the static policy allocates CPUs using a topology-aware best-fit allocation. On SMT enabled systems, the policy can allocate individual virtual cores, which correspond to hardware threads. This can lead to different containers sharing the same physical cores; this behaviour in turn contributes to the [noisy neighbours problem](#). With the option enabled, the pod will be admitted by the kubelet only if the CPU request of all its containers can be fulfilled by allocating full physical cores. If the pod does not pass the admission, it will be put in Failed state with the message `SMTAlignmentError`.

If the `distribute-cpus-across-numa` policy option is specified, the static policy will evenly distribute CPUs across NUMA nodes in cases where more than one NUMA node is required to satisfy the allocation. By default, the `CPUManager` will pack CPUs onto one NUMA node until it is filled, with any remaining CPUs simply spilling over to the next NUMA node. This can cause undesired bottlenecks in parallel code relying on barriers (and similar synchronization primitives), as this type of code tends to run only as fast as its slowest worker (which is slowed down by the fact that fewer CPUs are available on at least one NUMA node). By distributing CPUs evenly across NUMA nodes, application developers can more easily ensure that no single worker suffers from NUMA effects more than any other, improving the overall performance of these types of applications.

If the `align-by-socket` policy option is specified, CPUs will be considered aligned at the socket boundary when deciding how to allocate CPUs to a container. By default, the `CPUManager` aligns CPU allocations at the NUMA boundary, which could result in performance degradation if CPUs need to be pulled from more than one NUMA node to satisfy the allocation. Although it tries to ensure that all CPUs are allocated from the *minimum* number of NUMA nodes, there is no guarantee that those NUMA nodes will be on the same socket. By directing the `CPUManager` to explicitly align CPUs at the socket boundary rather than the NUMA boundary, we are able to avoid such issues. Note, this policy option is not compatible with `TopologyManager single-numa-node` policy and does not apply to hardware where the number of sockets is greater than number of NUMA nodes.

The `full-pcpus-only` option can be enabled by adding `full-pcpus-only=true` to the `CPUManager` policy options. Likewise, the `distribute-cpus-across-numa` option can be enabled by adding `distribute-cpus-across-`

`numa=true` to the CPUManager policy options. When both are set, they are "additive" in the sense that CPUs will be distributed across NUMA nodes in chunks of full-ppcups rather than individual cores. The `align-by-socket` policy option can be enabled by adding `align-by-socket=true` to the CPUManager policy options. It is also additive to the `full-ppcups-only` and `distribute-ccpus-across-numa` policy options.

2.16 - Control Topology Management Policies on a node

FEATURE STATE: [Kubernetes v1.27 \[stable\]](#)

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

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

Topology Manager is a Kubelet component that aims to coordinate the set of components that are responsible for these optimizations.

Before you begin

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

- [KillerCoda](#)
- [Play with Kubernetes](#)

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

How Topology Manager Works

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

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

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

The Topology manager receives Topology information from the *Hint Providers* as a bitmask denoting NUMA Nodes available and a preferred allocation indication. The Topology Manager policies perform a set of operations on the hints provided and converge on the hint determined by the policy to give the optimal result, if an undesirable hint is stored the preferred field for the hint will be set to false. In the current policies preferred is the narrowest preferred mask. The selected hint is stored as

part of the Topology Manager. Depending on the policy configured the pod can be accepted or rejected from the node based on the selected hint. The hint is then stored in the Topology Manager for use by the *Hint Providers* when making the resource allocation decisions.

Topology Manager Scopes and Policies

The Topology Manager currently:

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

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

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

The `scope` defines the granularity at which you would like resource alignment to be performed (e.g. at the `pod` or `container` level). And the `policy` defines the actual strategy used to carry out the alignment (e.g. `best-effort`, `restricted`, `single-numa-node`, etc.). Details on the various scopes and policies available today can be found below.

Note: To align CPU resources with other requested resources in a Pod Spec, the CPU Manager should be enabled and proper CPU Manager policy should be configured on a Node. See [control CPU Management Policies](#).

Note: To align memory (and hugepages) resources with other requested resources in a Pod Spec, the Memory Manager should be enabled and proper Memory Manager policy should be configured on a Node. Examine [Memory Manager](#) documentation.

Topology Manager Scopes

The Topology Manager can deal with the alignment of resources in a couple of distinct scopes:

- `container` (default)
- `pod`

Either option can be selected at a time of the kubelet startup, with `--topology-manager-scope` flag.

container scope

The `container` scope is used by default.

Within this scope, the Topology Manager performs a number of sequential resource alignments, i.e., for each container (in a pod) a separate alignment is computed. In other words, there is no notion of grouping the containers to a specific set of NUMA nodes, for this particular scope. In effect, the Topology Manager performs an arbitrary alignment of individual containers to NUMA nodes.

The notion of grouping the containers was endorsed and implemented on purpose in the following scope, for example the `pod` scope.

pod scope

To select the `pod` scope, start the kubelet with the command line option `--topology-manager-scope=pod`.

This scope allows for grouping all containers in a pod to a common set of NUMA nodes. That is, the Topology Manager treats a pod as a whole and attempts to allocate the entire pod (all containers) to either a single NUMA node or a common set of NUMA nodes. The following examples illustrate the alignments produced by the Topology Manager on different occasions:

- all containers can be and are allocated to a single NUMA node;
- all containers can be and are allocated to a shared set of NUMA nodes.

The total amount of particular resource demanded for the entire pod is calculated according to [effective requests/limits](#) formula, and thus, this total value is equal to the maximum of:

- the sum of all app container requests,
- the maximum of init container requests,

for a resource.

Using the `pod` scope in tandem with `single-numa-node` Topology Manager policy is specifically valuable for workloads that are latency sensitive or for high-throughput applications that perform IPC. By combining both options, you are able to place all containers in a pod onto a single NUMA node; hence, the inter-NUMA communication overhead can be eliminated for that pod.

In the case of `single-numa-node` policy, a pod is accepted only if a suitable set of NUMA nodes is present among possible allocations. Reconsider the example above:

- a set containing only a single NUMA node - it leads to pod being admitted,
- whereas a set containing more NUMA nodes - it results in pod rejection (because instead of one NUMA node, two or more NUMA nodes are required to satisfy the allocation).

To recap, Topology Manager first computes a set of NUMA nodes and then tests it against Topology Manager policy, which either leads to the rejection or admission of the pod.

Topology Manager Policies

Topology Manager supports four allocation policies. You can set a policy via a Kubelet flag, `--topology-manager-policy`. There are four supported policies:

- `none` (default)
- `best-effort`
- `restricted`
- `single-numa-node`

Note: If Topology Manager is configured with the `pod` scope, the container, which is considered by the policy, is reflecting requirements of the entire pod, and thus each container from the pod will result with **the same** topology alignment decision.

none policy

This is the default policy and does not perform any topology alignment.

best-effort policy

For each container in a Pod, the kubelet, with `best-effort` topology management policy, calls each Hint Provider to discover their resource availability. Using this information, the Topology Manager stores the preferred NUMA Node affinity for that container. If the affinity is not preferred, Topology Manager will store this and admit the pod to the node anyway.

The *Hint Providers* can then use this information when making the resource allocation decision.

restricted policy

For each container in a Pod, the kubelet, with `restricted` topology management policy, calls each Hint Provider to discover their resource availability. Using this information, the Topology Manager stores the preferred NUMA Node affinity for that container. If the affinity is not preferred, Topology Manager will reject this pod from the node. This will result in a pod in a `Terminated` state with a pod admission failure.

Once the pod is in a `Terminated` state, the Kubernetes scheduler will **not** attempt to reschedule the pod. It is recommended to use a ReplicaSet or Deployment to trigger a redeploy of the pod. An external control loop could be also implemented to trigger a redeployment of pods that have the `Topology Affinity` error.

If the pod is admitted, the *Hint Providers* can then use this information when making the resource allocation decision.

single-numa-node policy

For each container in a Pod, the kubelet, with `single-numa-node` topology management policy, calls each Hint Provider to discover their resource availability. Using this information, the Topology Manager determines if a single NUMA Node affinity is possible. If it is, Topology Manager will store this and the *Hint Providers* can then use this information when making the resource allocation decision. If, however, this is not possible then the Topology Manager will reject the pod from the node. This will result in a pod in a `Terminated` state with a pod admission failure.

Once the pod is in a `Terminated` state, the Kubernetes scheduler will **not** attempt to reschedule the pod. It is recommended to use a Deployment with replicas to trigger a redeploy of the Pod. An external control loop could be also implemented to trigger a redeployment of pods that have the `Topology Affinity` error.

Topology manager policy options

Support for the Topology Manager policy options requires `TopologyManagerPolicyOptions` [feature gate](#) to be enabled (it is enabled by default).

You can toggle groups of options on and off based upon their maturity level using the following feature gates:

- `TopologyManagerPolicyBetaOptions` default enabled. Enable to show beta-level options.
- `TopologyManagerPolicyAlphaOptions` default disabled. Enable to show alpha-level options.

You will still have to enable each option using the `TopologyManagerPolicyOptions` kubelet option.

The following policy options exists:

- `prefer-closest-numa-nodes` (beta, visible by default; `TopologyManagerPolicyOptions` and `TopologyManagerPolicyBetaOptions` feature gates have to be enabled). The `prefer-closest-numa-nodes` policy option is beta in Kubernetes 1.28.

If the `prefer-closest-numa-nodes` policy option is specified, the `best-effort` and `restricted` policies will favor sets of NUMA nodes with shorter distance between them when making admission decisions. You can enable this option by adding `prefer-closest-numa-nodes=true` to the Topology Manager policy options. By default, without this option, Topology Manager aligns resources on either a single NUMA node or the minimum number of NUMA nodes (in cases where more than one NUMA node is required). However, the `TopologyManager` is not aware of NUMA distances and does not take them into account when making admission decisions. This limitation surfaces in multi-socket, as well as single-socket multi NUMA systems, and can cause significant performance degradation in latency-critical execution and high-throughput applications if the Topology Manager decides to align resources on non-adjacent NUMA nodes.

Pod Interactions with Topology Manager Policies

Consider the containers in the following pod specs:

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

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

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

This pod runs in the `Burstable` QoS class because `requests` are less than `limits`.

If the selected policy is anything other than `none`, Topology Manager would consider these Pod specifications. The Topology Manager would consult the Hint Providers to get topology hints. In the case of the `static`, the CPU Manager policy would return default topology hint, because these Pods do not have explicitly request CPU resources.

```
spec:
  containers:
    - name: nginx
      image: nginx
      resources:
        limits:
          memory: "200Mi"
          cpu: "2"
          example.com/device: "1"
        requests:
          memory: "200Mi"
          cpu: "2"
          example.com/device: "1"
```

This pod with integer CPU request runs in the `Guaranteed` QoS class because `requests` are equal to `limits`.

```
spec:
  containers:
    - name: nginx
      image: nginx
      resources:
        limits:
          memory: "200Mi"
          cpu: "300m"
          example.com/device: "1"
        requests:
          memory: "200Mi"
          cpu: "300m"
          example.com/device: "1"
```

This pod with sharing CPU request runs in the `Guaranteed` QoS class because `requests` are equal to `limits`.

```
spec:
  containers:
    - name: nginx
      image: nginx
      resources:
        limits:
          example.com/deviceA: "1"
          example.com/deviceB: "1"
        requests:
          example.com/deviceA: "1"
          example.com/deviceB: "1"
```

This pod runs in the `BestEffort` QoS class because there are no CPU and memory requests.

The Topology Manager would consider the above pods. The Topology Manager would consult the Hint Providers, which are CPU and Device Manager to get topology hints for the pods.

In the case of the `Guaranteed` pod with integer CPU request, the `static` CPU Manager policy would return topology hints relating to the exclusive CPU and the Device Manager would send back hints for the requested device.

In the case of the `Guaranteed` pod with sharing CPU request, the `static` CPU Manager policy would return default topology hint as there is no exclusive CPU request and the Device Manager would send back hints for the requested device.

In the above two cases of the `Guaranteed` pod, the `none` CPU Manager policy would return default topology hint.

In the case of the `BestEffort` pod, the `static` CPU Manager policy would send back the default topology hint as there is no CPU request and the Device Manager would send back the hints for each of the requested devices.

Using this information the Topology Manager calculates the optimal hint for the pod and stores this information, which will be used by the Hint Providers when they are making their resource assignments.

Known Limitations

1. The maximum number of NUMA nodes that Topology Manager allows is 8. With more than 8 NUMA nodes there will be a state explosion when trying to enumerate the possible NUMA affinities and generating their hints.
2. The scheduler is not topology-aware, so it is possible to be scheduled on a node and then fail on the node due to the Topology Manager.

2.17 - Customizing DNS Service

This page explains how to configure your DNS Pod(s) and customize the DNS resolution process in your cluster.

Before you begin

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

- [Killercoda](#)
- [Play with Kubernetes](#)

Your cluster must be running the CoreDNS add-on.

Your Kubernetes server must be at or later than version v1.12. To check the version, enter `kubectl version`.

Introduction

DNS is a built-in Kubernetes service launched automatically using the *addon manager* [cluster add-on](#).

Note: The CoreDNS Service is named `kube-dns` in the `metadata.name` field.

The intent is to ensure greater interoperability with workloads that relied on the legacy `kube-dns` Service name to resolve addresses internal to the cluster. Using a Service named `kube-dns` abstracts away the implementation detail of which DNS provider is running behind that common name.

If you are running CoreDNS as a Deployment, it will typically be exposed as a Kubernetes Service with a static IP address. The kubelet passes DNS resolver information to each container with the `--cluster-dns=<dns-service-ip>` flag.

DNS names also need domains. You configure the local domain in the kubelet with the flag `--cluster-domain=<default-local-domain>`.

The DNS server supports forward lookups (A and AAAA records), port lookups (SRV records), reverse IP address lookups (PTR records), and more. For more information, see [DNS for Services and Pods](#).

If a Pod's `dnsPolicy` is set to `default`, it inherits the name resolution configuration from the node that the Pod runs on. The Pod's DNS resolution should behave the same as the node. But see [Known issues](#).

If you don't want this, or if you want a different DNS config for pods, you can use the kubelet's `--resolv-conf` flag. Set this flag to `""` to prevent Pods from inheriting DNS. Set it to a valid file path to specify a file other than `/etc/resolv.conf` for DNS inheritance.

CoreDNS

CoreDNS is a general-purpose authoritative DNS server that can serve as cluster DNS, complying with the [DNS specifications](#).

CoreDNS ConfigMap options

CoreDNS is a DNS server that is modular and pluggable, with plugins adding new functionalities. The CoreDNS server can be configured by maintaining a [Corefile](#), which is the CoreDNS configuration file. As a cluster administrator, you can modify the [ConfigMap](#) for the CoreDNS Corefile to change how DNS service discovery behaves for that cluster.

In Kubernetes, CoreDNS is installed with the following default Corefile configuration:

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: coredns
  namespace: kube-system
data:
  Corefile: |
    .:53 {
      errors
      health {
        Lameduck 5s
      }
      ready
      kubernetes cluster.Local in-addr.arpa ip6.arpa {
        pods insecure
        fallthrough in-addr.arpa ip6.arpa
        ttl 30
      }
      prometheus :9153
      forward . /etc/resolv.conf
      cache 30
      Loop
      reload
      Loadbalance
    }
```

The Corefile configuration includes the following [plugins](#) of CoreDNS:

- [errors](#): Errors are logged to stdout.
- [health](#): Health of CoreDNS is reported to `http://localhost:8080/health`. In this extended syntax `lameduck` will make the process unhealthy then wait for 5 seconds before the process is shut down.
- [ready](#): An HTTP endpoint on port 8181 will return 200 OK, when all plugins that are able to signal readiness have done so.
- [kubernetes](#): CoreDNS will reply to DNS queries based on IP of the Services and Pods. You can find [more details](#) about this plugin on the CoreDNS website.
 - `ttl` allows you to set a custom TTL for responses. The default is 5 seconds. The minimum TTL allowed is 0 seconds, and the maximum is capped at 3600 seconds. Setting TTL to 0 will prevent records from being cached.
 - The `pods insecure` option is provided for backward compatibility with `kube-dns`.

- You can use the `pods verified` option, which returns an A record only if there exists a pod in the same namespace with a matching IP.
- The `pods disabled` option can be used if you don't use pod records.
- [prometheus](#): Metrics of CoreDNS are available at `http://localhost:9153/metrics` in the [Prometheus](#) format (also known as OpenMetrics).
- [forward](#): Any queries that are not within the Kubernetes cluster domain are forwarded to predefined resolvers (`/etc/resolv.conf`).
- [cache](#): This enables a frontend cache.
- [loop](#): Detects simple forwarding loops and halts the CoreDNS process if a loop is found.
- [reload](#): Allows automatic reload of a changed Corefile. After you edit the ConfigMap configuration, allow two minutes for your changes to take effect.
- [loadbalance](#): This is a round-robin DNS loadbalancer that randomizes the order of A, AAAA, and MX records in the answer.

You can modify the default CoreDNS behavior by modifying the ConfigMap.

Configuration of Stub-domain and upstream nameserver using CoreDNS

CoreDNS has the ability to configure stub-domains and upstream nameservers using the [forward plugin](#).

Example

If a cluster operator has a [Consul](#) domain server located at "10.150.0.1", and all Consul names have the suffix ".consul.local". To configure it in CoreDNS, the cluster administrator creates the following stanza in the CoreDNS ConfigMap.

```
consul.local:53 {
    errors
    cache 30
    forward . 10.150.0.1
}
```

To explicitly force all non-cluster DNS lookups to go through a specific nameserver at 172.16.0.1, point the `forward` to the nameserver instead of `/etc/resolv.conf`

```
forward . 172.16.0.1
```

The final ConfigMap along with the default `Corefile` configuration looks like:

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: coredns
  namespace: kube-system
data:
  Corefile: |
    .:53 {
      errors
      health
      kubernetes cluster.local in-addr.arpa ip6.arpa {
        pods insecure
        fallthrough in-addr.arpa ip6.arpa
      }
      prometheus :9153
      forward . 172.16.0.1
      cache 30
      loop
      reload
      loadbalance
    }
    consul.local:53 {
      errors
      cache 30
      forward . 10.150.0.1
    }
}
```

Note: CoreDNS does not support FQDNs for stub-domains and nameservers (eg: "ns.foo.com"). During translation, all FQDN nameservers will be omitted from the CoreDNS config.

What's next

- Read [Debugging DNS Resolution](#)

2.18 - Debugging DNS Resolution

This page provides hints on diagnosing DNS problems.

Before you begin

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

- [Killercoda](#)
- [Play with Kubernetes](#)

Your cluster must be configured to use the CoreDNS addon or its precursor, kube-dns.

Your Kubernetes server must be at or later than version v1.6. To check the version, enter `kubectl version`.

Create a simple Pod to use as a test environment

[admin/dns/dnsutils.yaml](#) 

```
apiVersion: v1
kind: Pod
metadata:
  name: dnsutils
  namespace: default
spec:
  containers:
    - name: dnsutils
      image: registry.k8s.io/e2e-test-images/jessie-dnsutils:1.3
      command:
        - sleep
        - "infinity"
      imagePullPolicy: IfNotPresent
  restartPolicy: Always
```

Note: This example creates a pod in the `default` namespace. DNS name resolution for services depends on the namespace of the pod. For more information, review [DNS for Services and Pods](#).

Use that manifest to create a Pod:

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

```
pod/dnsutils created
```

...and verify its status:

```
kubectl get pods dnsutils
```

NAME	READY	STATUS	RESTARTS	AGE
dnsutils	1/1	Running	0	<some-time>

Once that Pod is running, you can exec `nslookup` in that environment. If you see something like the following, DNS is working correctly.

```
kubectl exec -i -t dnsutils -- nslookup kubernetes.default
```

```
Server: 10.0.0.10
Address 1: 10.0.0.10

Name: kubernetes.default
Address 1: 10.0.0.1
```

If the `nslookup` command fails, check the following:

Check the local DNS configuration first

Take a look inside the `resolv.conf` file. (See [Customizing DNS Service](#) and [Known issues](#) below for more information)

```
kubectl exec -ti dnsutils -- cat /etc/resolv.conf
```

Verify that the search path and name server are set up like the following (note that search path may vary for different cloud providers):

```
search default.svc.cluster.local svc.cluster.local cluster.local google
nameserver 10.0.0.10
options ndots:5
```

Errors such as the following indicate a problem with the CoreDNS (or kube-dns) add-on or with associated Services:

```
kubectl exec -i -t dnsutils -- nslookup kubernetes.default
```

```
Server: 10.0.0.10
Address 1: 10.0.0.10

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

or

```
kubectl exec -i -t dnsutils -- nslookup kubernetes.default
```

```
Server: 10.0.0.10
Address 1: 10.0.0.10 kube-dns.kube-system.svc.cluster.local

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

Check if the DNS pod is running

Use the `kubectl get pods` command to verify that the DNS pod is running.

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

NAME	READY	STATUS	RESTARTS	AGE
...				
coredns-7b96bf9f76-5hsxb	1/1	Running	0	1h
coredns-7b96bf9f76-mvmmmt	1/1	Running	0	1h
...				

Note: The value for label `k8s-app` is `kube-dns` for both CoreDNS and `kube-dns` deployments.

If you see that no CoreDNS Pod is running or that the Pod has failed/completed, the DNS add-on may not be deployed by default in your current environment and you will have to deploy it manually.

Check for errors in the DNS pod

Use the `kubectl logs` command to see logs for the DNS containers.

For CoreDNS:

```
kubectl logs --namespace=kube-system -l k8s-app=kube-dns
```

Here is an example of a healthy CoreDNS log:

```
.:53
2018/08/15 14:37:17 [INFO] CoreDNS-1.2.2
2018/08/15 14:37:17 [INFO] linux/amd64, go1.10.3, 2e322f6
CoreDNS-1.2.2
linux/amd64, go1.10.3, 2e322f6
2018/08/15 14:37:17 [INFO] plugin/reload: Running configuration MD5 = 2
```

See if there are any suspicious or unexpected messages in the logs.

Is DNS service up?

Verify that the DNS service is up by using the `kubectl get service` command.

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

NAME	TYPE	CLUSTER-IP	EXTERNAL-IP	PORT(S)
...				
kube-dns	ClusterIP	10.0.0.10	<none>	53/UDP,53/TCP
...				

Note: The service name is `kube-dns` for both CoreDNS and kube-dns deployments.

If you have created the Service or in the case it should be created by default but it does not appear, see [debugging Services](#) for more information.

Are DNS endpoints exposed?

You can verify that DNS endpoints are exposed by using the `kubectl get endpoints` command.

```
kubectl get endpoints kube-dns --namespace=kube-system
```

NAME	ENDPOINTS	AGE
kube-dns	10.180.3.17:53,10.180.3.17:53	1h

If you do not see the endpoints, see the endpoints section in the [debugging Services](#) documentation.

For additional Kubernetes DNS examples, see the [cluster-dns examples](#) in the Kubernetes GitHub repository.

Are DNS queries being received/processed?

You can verify if queries are being received by CoreDNS by adding the `log` plugin to the CoreDNS configuration (aka Corefile). The CoreDNS Corefile is held in a ConfigMap named `coredns`. To edit it, use the command:

```
kubectl -n kube-system edit configmap coredns
```

Then add `log` in the Corefile section per the example below:

```

apiVersion: v1
kind: ConfigMap
metadata:
  name: coredns
  namespace: kube-system
data:
  Corefile: |
    .:53 {
      Log
      errors
      health
      kubernetes cluster.local in-addr.arpa ip6.arpa {
        pods insecure
        upstream
        fallthrough in-addr.arpa ip6.arpa
      }
      prometheus :9153
      forward . /etc/resolv.conf
      cache 30
      Loop
      reload
      Loadbalance
    }
}

```

After saving the changes, it may take up to minute or two for Kubernetes to propagate these changes to the CoreDNS pods.

Next, make some queries and view the logs per the sections above in this document. If CoreDNS pods are receiving the queries, you should see them in the logs.

Here is an example of a query in the log:

```

.:53
2018/08/15 14:37:15 [INFO] CoreDNS-1.2.0
2018/08/15 14:37:15 [INFO] linux/amd64, go1.10.3, 2e322f6
CoreDNS-1.2.0
linux/amd64, go1.10.3, 2e322f6
2018/09/07 15:29:04 [INFO] plugin/reload: Running configuration MD5 = 1
2018/09/07 15:29:04 [INFO] Reloading complete
172.17.0.18:41675 - [07/Sep/2018:15:29:11 +0000] 59925 "A IN kubernetes

```

Does CoreDNS have sufficient permissions?

CoreDNS must be able to list `service` and `endpoint` related resources to properly resolve service names.

Sample error message:

```
2022-03-18T07:12:15.699431183Z [INFO] 10.96.144.227:52299 - 3686 "A IN
```

First, get the current ClusterRole of `system:coredns`:

```
kubectl describe clusterrole system:coredns -n kube-system
```

Expected output:

PolicyRule:	Resources	Non-Resource URLs	Resource Names	Verb
	-----	-----	-----	-----
endpoints	[]	[]	[]	
namespaces	[]	[]	[]	
pods	[]	[]	[]	
services	[]	[]	[]	
endpointslices.discovery.k8s.io	[]	[]	[]	[]

If any permissions are missing, edit the ClusterRole to add them:

```
kubectl edit clusterrole system:coredns -n kube-system
```

Example insertion of EndpointSlices permissions:

```
...
- apiGroups:
  - discovery.k8s.io
resources:
- endpointslices
verbs:
- list
- watch
...
```

Are you in the right namespace for the service?

DNS queries that don't specify a namespace are limited to the pod's namespace.

If the namespace of the pod and service differ, the DNS query must include the namespace of the service.

This query is limited to the pod's namespace:

```
kubectl exec -i -t dnsutils -- nslookup <service-name>
```

This query specifies the namespace:

```
kubectl exec -i -t dnsutils -- nslookup <service-name>.<namespace>
```

To learn more about name resolution, see [DNS for Services and Pods](#).

Known issues

Some Linux distributions (e.g. Ubuntu) use a local DNS resolver by default (`systemd-resolved`). `Systemd-resolved` moves and replaces `/etc/resolv.conf` with a stub file that can cause a fatal forwarding loop when resolving names in upstream servers. This can be fixed manually by using `kubelet`'s `--resolv-conf` flag to point to the correct `resolv.conf` (With `systemd-resolved`, this is `/run/systemd/resolve/resolv.conf`). `kubeadm` automatically detects `systemd-resolved`, and adjusts the `kubelet` flags accordingly.

Kubernetes installs do not configure the nodes' `resolv.conf` files to use the cluster DNS by default, because that process is inherently distribution-specific. This should probably be implemented eventually.

Linux's libc (a.k.a. glibc) has a limit for the DNS `nameserver` records to 3 by default and Kubernetes needs to consume 1 `nameserver` record. This means that if a local installation already uses 3 `nameserver`s, some of those entries will be lost. To work around this limit, the node can run `dnsmasq`, which will provide more `nameserver` entries. You can also use kubelet's `--resolv-conf` flag.

If you are using Alpine version 3.17 or earlier as your base image, DNS may not work properly due to a design issue with Alpine. Until musl version 1.24 didn't include TCP fallback to the DNS stub resolver meaning any DNS call above 512 bytes would fail. Please upgrade your images to Alpine version 3.18 or above.

What's next

- See [Autoscaling the DNS Service in a Cluster](#).
- Read [DNS for Services and Pods](#)

2.19 - 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.

Note: This section links to third party projects that provide functionality required by Kubernetes. The Kubernetes project authors aren't responsible for these projects, which are listed alphabetically. To add a project to this list, read the [content guide](#) before submitting a change. [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. It is recommended to run this tutorial on a cluster with at least two nodes that are not acting as control plane hosts. If you do not already have a cluster, you can create one by using [minikube](#) or you can use one of these Kubernetes playgrounds:

- [Killercoda](#)
- [Play with Kubernetes](#)

Your Kubernetes server must be at or later than version v1.8. To check the version, enter `kubectl version`.

Make sure you've configured a network provider with network policy support. There are a number of network providers that support `NetworkPolicy`, including:

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

Create an `nginx` deployment and expose it via a service

To see how Kubernetes network policy works, start off by creating an `nginx` Deployment.

```
kubectl create deployment nginx --image=nginx
```

```
deployment.apps/nginx created
```

Expose the Deployment through a Service called `nginx`.

```
kubectl expose deployment nginx --port=80
```

```
service/nginx exposed
```

The above commands create a Deployment with an nginx Pod and expose the Deployment through a Service named `nginx`. The `nginx` Pod and Deployment are found in the `default` namespace.

```
kubectl get svc,pod
```

NAME	CLUSTER-IP	EXTERNAL-IP	PORT(S)	AGE
service/kubernetes	10.100.0.1	<none>	443/TCP	46m
service/nginx	10.100.0.16	<none>	80/TCP	33s
NAME	READY	STATUS	RESTARTS	AGE
pod/nginx-701339712-e0qfq	1/1	Running	0	35s

Test the service by accessing it from another Pod

You should be able to access the new `nginx` service from other Pods. To access the `nginx` Service from another Pod in the `default` namespace, start a busybox container:

```
kubectl run busybox --rm -ti --image=busybox:1.28 -- /bin/sh
```

In your shell, run the following command:

```
wget --spider --timeout=1 nginx
```

```
Connecting to nginx (10.100.0.16:80)
remote file exists
```

Limit access to the `nginx` service

To limit the access to the `nginx` service so that only Pods with the label `access: true` can query it, create a NetworkPolicy object as follows:

[service/networking/nginx-policy.yaml](#) 

```
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: access-nginx
spec:
  podSelector:
    matchLabels:
      app: nginx
  ingress:
  - from:
    - podSelector:
        matchLabels:
          access: "true"
```

The name of a NetworkPolicy object must be a valid [DNS subdomain name](#).

Note: NetworkPolicy includes a `podSelector` which selects the grouping of Pods to which the policy applies. You can see this policy selects Pods with the label `app=nginx`. The label was automatically added to the Pod in the `nginx` Deployment. An empty `podSelector` selects all pods in the namespace.

Assign the policy to the service

Use kubectl to create a NetworkPolicy from the above `nginx-policy.yaml` file:

```
kubectl apply -f https://k8s.io/examples/service/networking/nginx-pol
```

```
networkpolicy.networking.k8s.io/access-nginx created
```

Test access to the service when access label is not defined

When you attempt to access the `nginx` Service from a Pod without the correct labels, the request times out:

```
kubectl run busybox --rm -ti --image=busybox:1.28 -- /bin/sh
```

In your shell, run the command:

```
wget --spider --timeout=1 nginx
```

```
Connecting to nginx (10.100.0.16:80)
wget: download timed out
```

Define access label and test again

You can create a Pod with the correct labels to see that the request is allowed:

```
kubectl run busybox --rm -ti --labels="access=true" --image=busybox:1.0
```

In your shell, run the command:

```
wget --spider --timeout=1 nginx
```

```
Connecting to nginx (10.100.0.16:80)
remote file exists
```

2.20 - Developing Cloud Controller Manager

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

The cloud-controller-manager is a Kubernetes [control plane](#) component that embeds cloud-specific control logic. The [cloud controller manager](#) lets you link your cluster into your cloud provider's API, and separates out the components that interact with that cloud platform from components that only interact with your cluster.

By decoupling the interoperability logic between Kubernetes and the underlying cloud infrastructure, the cloud-controller-manager component enables cloud providers to release features at a different pace compared to the main Kubernetes project.

Background

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

The Kubernetes project provides skeleton cloud-controller-manager code with Go interfaces to allow you (or your cloud provider) to plug in your own implementations. This means that a cloud provider can implement a cloud-controller-manager by importing packages from Kubernetes core; each cloudprovider will register their own code by calling `cloudprovider.RegisterCloudProvider` to update a global variable of available cloud providers.

Developing

Out of tree

To build an out-of-tree cloud-controller-manager for your cloud:

1. Create a go package with an implementation that satisfies [cloudprovider.Interface](#).
2. Use [main.go in cloud-controller-manager](#) from Kubernetes core as a template for your `main.go`. As mentioned above, the only difference should be the cloud package that will be imported.
3. Import your cloud package in `main.go`, ensure your package has an `init` block to run [cloudprovider.RegisterCloudProvider](#).

Many cloud providers publish their controller manager code as open source. If you are creating a new cloud-controller-manager from scratch, you could take an existing out-of-tree cloud controller manager as your starting point.

In tree

For in-tree cloud providers, you can run the in-tree cloud controller manager as a [DaemonSet](#) in your cluster. See [Cloud Controller Manager Administration](#) for more details.

2.21 - Enable Or Disable A Kubernetes API

This page shows how to enable or disable an API version from your cluster's control plane.

Specific API versions can be turned on or off by passing `--runtime-config=api/<version>` as a command line argument to the API server. The values for this argument are a comma-separated list of API versions. Later values override earlier values.

The `runtime-config` command line argument also supports 2 special keys:

- `api/all` , representing all known APIs
- `api/legacy` , representing only legacy APIs. Legacy APIs are any APIs that have been explicitly [deprecated](#).

For example, to turn off all API versions except v1, pass `--runtime-config=api/all=false,api/v1=true` to the `kube-apiserver` .

What's next

Read the [full documentation](#) for the `kube-apiserver` component.

2.22 - Encrypting Confidential Data at Rest

All of the APIs in Kubernetes that let you write persistent API resource data support at-rest encryption. For example, you can enable at-rest encryption for [Secrets](#). This at-rest encryption is additional to any system-level encryption for the etcd cluster or for the filesystem(s) on hosts where you are running the kube-apiserver.

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

Note:

This task covers encryption for resource data stored using the [Kubernetes API](#). For example, you can encrypt Secret objects, including the key-value data they contain.

If you want to encrypt data in filesystems that are mounted into containers, you instead need to either:

- use a storage integration that provides encrypted [volumes](#)
- encrypt the data within your own 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. It is recommended to run this tutorial on a cluster with at least two nodes that are not acting as control plane hosts. If you do not already have a cluster, you can create one by using [minikube](#) or you can use one of these Kubernetes playgrounds:
 - [Killercoda](#)
 - [Play with Kubernetes](#)
- This task assumes that you are running the Kubernetes API server as a [static pod](#) on each control plane node.
- Your cluster's control plane **must** use etcd v3.x (major version 3, any minor version).
- To encrypt a custom resource, your cluster must be running Kubernetes v1.26 or newer.
- To use a wildcard to match resources, your cluster must be running Kubernetes v1.27 or newer.

To check the version, enter `kubectl version`.

Configuration and determining whether encryption at rest is already enabled

The `kube-apiserver` process accepts an argument `--encryption-provider-config` that controls how API data is encrypted in etcd. The configuration is provided as an API named [EncryptionConfiguration](#). An example configuration is provided below.

Caution: IMPORTANT: For high-availability configurations (with two or more control plane nodes), the encryption configuration file must be the same! Otherwise, the `kube-apiserver` component cannot decrypt data stored in the etcd.

Understanding the encryption at rest configuration

```
---  
#  
# CAUTION: this is an example configuration.  
#           Do not use this for your own cluster!  
  
#  
apiVersion: apiserver.config.k8s.io/v1  
kind: EncryptionConfiguration  
  
resources:  
  - resources:  
    - secrets  
    - configmaps  
    - pandas.awesome.bears.example # a custom resource API  
  
providers:  
  # This configuration does not provide data confidentiality. The j  
  # configured provider is specifying the "identity" mechanism, wh  
  # stores resources as plain text.  
  #  
  - identity: {} # plain text, in other words NO encryption  
  - aesgcm:  
    keys:  
      - name: key1  
        secret: c2VjcmV0IGlzIHNlY3VyZQ==  
      - name: key2  
        secret: dGhpcyBpcyBwYXNzd29yZA==  
  - aescbc:  
    keys:  
      - name: key1  
        secret: c2VjcmV0IGlzIHNlY3VyZQ==  
      - name: key2  
        secret: dGhpcyBpcyBwYXNzd29yZA==  
  - secretbox:  
    keys:  
      - name: key1  
        secret: YWJjZGVmZ2hpamtsbW5vcHFyc3R1dnd4eXoxMjM0NTY=  
  
  - resources:  
    - events  
  
providers:  
  - identity: {} # do not encrypt Events even though *.* is specif  
  
resources:  
  - '*.apps' # wildcard match requires Kubernetes 1.27 or later  
  
providers:  
  - aescbc:  
    keys:  
      - name: key2  
        secret: c2VjcmV0IGlzIHNlY3VyZSwgb3IgaXMgaXQ/Cg==  
  
resources:  
  - '*.*' # wildcard match requires Kubernetes 1.27 or later  
  
providers:  
  - aescbc:  
    keys:  
      - name: key3  
        secret: c2VjcmV0IGlzIHNlY3VyZSwgSSB0aGluaw==
```

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 like Secrets, ConfigMaps, or other resources.

If custom resources are added to `EncryptionConfiguration` and the cluster version is 1.26 or newer, any newly created custom resources mentioned in the `EncryptionConfiguration` will be encrypted. Any custom resources that existed in etcd prior to that version and configuration will be unencrypted until they are next written to storage. This is the same behavior as built-in resources. See the [Ensure all secrets are encrypted](#) section.

The `providers` array is an ordered list of the possible encryption providers to use for the APIs that you listed. 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.

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 written into the storage. When reading resources from storage, each provider that matches the stored data attempts in order to decrypt the data. 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.

`EncryptionConfiguration` supports the use of wildcards to specify the resources that should be encrypted. Use '`*.<group>`' to encrypt all resources within a group (for eg '`*.apps`' in above example) or '`*.*`' to encrypt all resources. '`*.`' can be used to encrypt all resource in the core group. '`*.*`' will encrypt all resources, even custom resources that are added after API server start.

Note: Use of wildcards that overlap within the same resource list or across multiple entries are not allowed since part of the configuration would be ineffective. The `resources` list's processing order and precedence are determined by the order it's listed in the configuration.

Opting out of encryption for specific resources while wildcard is enabled can be achieved by adding a new `resources` array item with the resource name, followed by the `providers` array item with the `identity` provider. For example, if '`*.*`' is enabled and you want to opt-out encryption for the `events` resource, add a new item to the `resources` array with `events` as the resource name, followed by the providers array item with `identity`. The new item should look like this:

```
- resources:  
  - events  
providers:  
  - identity: {}
```

Ensure that the new item is listed before the wildcard '`*.*`' item in the `resources` array to give it precedence.

For more detailed information about the `EncryptionConfiguration` struct, please refer to the [encryption configuration API](#).

Caution: 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.

Available providers

Before you configure encryption-at-rest for data in your cluster's Kubernetes API, you need to select which provider(s) you will use.

The following table describes each available provider.

Name	Encryption	Strength	Speed	Key length
	None	N/A	N/A	N/A
identity	Resources written as-is without encryption. When set as the first provider, the resource will be decrypted as new values are written. Existing encrypted resources are not automatically overwritten with the plaintext data. The <code>identity</code> provider is the default if you do not specify otherwise.			
aescbc	AES-CBC with PKCS#7 padding	Weak	Fast	32-byte
	Not recommended due to CBC's vulnerability to padding oracle attacks. Key material accessible from control plane host.			
aesgcm	AES-GCM with random nonce	Must be rotated every 200,000 writes	Fastest	16, 24, or 32-byte
	Not recommended for use except when an automated key rotation scheme is implemented. Key material accessible from control plane host.			
kms v1 <i>(deprecated since Kubernetes v1.28)</i>	Uses envelope encryption scheme with DEK per resource.	Strongest	Slow (compared to <i>kms</i> version 2)	32-bytes
	Data is encrypted by data encryption keys (DEKs) using AES-GCM; DEKs are encrypted by key encryption keys (KEKs) according to configuration in Key Management Service (KMS). Simple key rotation, with a new DEK generated for each encryption, and KEK rotation controlled by the user. Read how to configure the KMS V1 provider .			

Name	Encryption	Strength	Speed	Key length
	Uses envelope encryption scheme with DEK per API server.	Strongest	Fast	32-bytes
kms v2 (beta)	Data is encrypted by data encryption keys (DEKs) using AES-GCM; DEKs are encrypted by key encryption keys (KEKs) according to configuration in Key Management Service (KMS). Kubernetes defaults to generating a new DEK at API server startup, which is then reused for object encryption. If you enable the KMSv2KDF feature gate , Kubernetes instead generates a new DEK per encryption from a secret seed. Whichever approach you configure, the DEK or seed is also rotated whenever the KEK is rotated. A good choice if using a third party tool for key management. Available in beta from Kubernetes v1.27. Read how to configure the KMS V2 provider .			

The `identity` provider is the default if you do not specify otherwise. **The identity provider does not encrypt stored data and provides no additional confidentiality protection.**

Key storage

Local key storage

Encrypting secret data with a locally managed key protects against an etcd compromise, but it fails to protect against a host compromise. Since the encryption keys are stored on the host in the `EncryptionConfiguration` YAML file, a skilled attacker can access that file and extract the encryption keys.

Managed (KMS) key storage

The KMS provider uses *envelope encryption*: Kubernetes encrypts resources using a data key, and then encrypts that data key using the managed encryption service. Kubernetes generates a unique data key for each resource. The API server stores an encrypted version of the data key in etcd alongside the ciphertext; when reading the resource, the API server calls the managed encryption service and provides both the ciphertext and the (encrypted) data key. Within the managed encryption service, the provider use a *key encryption key* to decipher the data key, deciphers the data key, and finally recovers the plain text. Communication between the control plane and the KMS requires in-transit protection, such as TLS.

Using envelope encryption creates dependence on the key encryption key, which is not stored in Kubernetes. In the KMS case, an attacker who intends to get unauthorised access to the plaintext values would need to compromise etcd **and** the third-party KMS provider.

Write an encryption configuration file

Caution: The encryption configuration file may contain keys that can decrypt content in etcd. If the configuration file contains any key material, you must properly restrict permissions on all your control plane hosts so only the user who runs the kube-apiserver can read this configuration.

Create a new encryption configuration file. The contents should be similar to:

```
---
```

```
apiVersion: apiserver.config.k8s.io/v1
kind: EncryptionConfiguration
resources:
  - resources:
    - secrets
    - configmaps
    - pandas.awesome.bears.example
  providers:
    - aescbc:
      keys:
        - name: key1
          # See the following text for more details about the secret
          secret: <BASE 64 ENCODED SECRET>
        - identity: {} # this fallback allows reading unencrypted secrets
                      # for example, during initial migration
```

To create a new Secret, perform the following steps:

1. Generate a 32-byte random key and base64 encode it. If you're on Linux or macOS, run the following command:

```
head -c 32 /dev/urandom | base64
```

2. Place that value in the `secret` field of the `EncryptionConfiguration` struct.
3. Set the `--encryption-provider-config` flag on the `kube-apiserver` to point to the location of the config file.

You will need to mount the new encryption config file to the `kube-apiserver` static pod. Here is an example on how to do that:

1. Save the new encryption config file to `/etc/kubernetes/enc/enc.yaml` on the control-plane node.
2. Edit the manifest for the `kube-apiserver` static pod: `/etc/kubernetes/manifests/kube-apiserver.yaml` similarly to this:

```

---
#
# This is a fragment of a manifest for a static Pod.
# Check whether this is correct for your cluster and for your API
#
apiVersion: v1
kind: Pod
metadata:
  annotations:
    kubeadm.kubernetes.io/kube-apiserver.advertise-address.endpoint: <IP>
  creationTimestamp: null
  labels:
    app.kubernetes.io/component: kube-apiserver
    tier: control-plane
    name: kube-apiserver
    namespace: kube-system
spec:
  containers:
    - command:
      - kube-apiserver
      ...
      - --encryption-provider-config=/etc/kubernetes/enc/enc.yaml
  volumeMounts:
    ...
    - name: enc                         # add this line
      mountPath: /etc/kubernetes/enc      # add this line
      readOnly: true                      # add this line
    ...
  volumes:
    ...
    - name: enc                         # add this line
      hostPath:
        path: /etc/kubernetes/enc        # add this line
        type: DirectoryOrCreate         # add this line
    ...

```

4. Restart your API server.

Caution: Your config file contains keys that can decrypt the contents in etcd, so you must properly restrict permissions on your control-plane nodes so only the user who runs the `kube-apiserver` can read it.

Reconfigure other control plane hosts

If you have multiple API servers in your cluster, you should deploy the changes in turn to each API server.

Make sure that you use the **same** encryption configuration on each control plane host.

Verify that newly written data is encrypted

Data is encrypted when written to etcd. After restarting your `kube-apiserver`, any newly created or updated Secret (or other resource kinds configured in `EncryptionConfiguration`) should be encrypted when stored.

To check this, you can use the `etcdctl` command line program to retrieve the contents of your secret data.

This example shows how to check this for encrypting the Secret API.

1. Create a new Secret called `secret1` in the `default` namespace:

```
kubectl create secret generic secret1 -n default --from-literal=m
```

- Using the `etcdctl` command line tool, read that Secret out of etcd:

```
ETCDCTL_API=3 etcdctl get /registry/secrets/default/secret1 [...]
```

where [...] must be the additional arguments for connecting to the etcd server.

For example:

```
ETCDCTL_API=3 etcdctl \
--cacert=/etc/kubernetes/pki/etcd/ca.crt \
--cert=/etc/kubernetes/pki/etcd/server.crt \
--key=/etc/kubernetes/pki/etcd/server.key \
get /registry/secrets/default/secret1 | hexdump -c
```

The output is similar to this (abbreviated):

```
00000000  2f 72 65 67 69 73 74 72  79 2f 73 65 63 72 65 74 |/reg
00000010  73 2f 64 65 66 61 75 6c  74 2f 73 65 63 72 65 74 |s/de
00000020  31 0a 6b 38 73 3a 65 6e  63 3a 61 65 73 63 62 63 |1.k8
00000030  3a 76 31 3a 6b 65 79 31  3a c7 6c e7 d3 09 bc 06 |:v1:
00000040  25 51 91 e4 e0 6c e5 b1  4d 7a 8b 3d b9 c2 7c 6e |%Q..
00000050  b4 79 df 05 28 ae 0d 8e  5f 35 13 2c c0 18 99 3e |.y..
[...]
00000110  23 3a 0d fc 28 ca 48 2d  6b 2d 46 cc 72 0b 70 4c |#:...
00000120  a5 fc 35 43 12 4e 60 ef  bf 6f fe cf df 0b ad 1f |..5C
00000130  82 c4 88 53 02 da 3e 66  ff 0a                      |...S
0000013a
```

- Verify the stored Secret is prefixed with `k8s:enc:aescbc:v1:` which indicates the `aescbc` provider has encrypted the resulting data. Confirm that the key name shown in `etcd` matches the key name specified in the `EncryptionConfiguration` mentioned above. In this example, you can see that the encryption key named `key1` is used in `etcd` and in `EncryptionConfiguration`.

- Verify the Secret is correctly decrypted when retrieved via the API:

```
kubectl get secret secret1 -n default -o yaml
```

The output should contain `mykey: bX1kYXRh`, with contents of `mydata` encoded using base64; read [decoding a Secret](#) to learn how to completely decode the Secret.

Ensure all relevant data are encrypted

It's often not enough to make sure that new objects get encrypted: you also want that encryption to apply to the objects that are already stored.

For this example, you have configured your cluster so that Secrets are encrypted on write. Performing a replace operation for each Secret will encrypt that content at rest, where the objects are unchanged.

You can make this change across all Secrets in your cluster:

```
# Run this as an administrator that can read and write all Secrets
kubectl get secrets --all-namespaces -o json | kubectl replace -f -
```

The command above reads all Secrets and then updates them with the same data, in order to apply server side encryption.

Note:

If an error occurs due to a conflicting write, retry the command. It is safe to run that command more than once.

For larger clusters, you may wish to subdivide the Secrets by namespace, or script an update.

Rotating a decryption key

Changing a 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 have backed up etcd with the new key in use and updated all Secrets

When running a single `kube-apiserver` instance, step 2 may be skipped.

Configure automatic reloading

You can configure automatic reloading of encryption provider configuration. That setting determines whether the `--encryption-provider-config` file you specify for `--encryption-provider-config` only once at startup, or automatically whenever you change that file. Enabling this option allows you to change the keys for encryption at rest without restarting the API server.

To allow automatic reloading, configure the API server to run with: `--encryption-provider-config-automatic-reload=true`

What's next

- Read about [decrypting data that are already stored at rest](#)
- Learn more about the [EncryptionConfiguration configuration API \(v1\)](#).

2.23 - Decrypt Confidential Data that is Already Encrypted at Rest

All of the APIs in Kubernetes that let you write persistent API resource data support at-rest encryption. For example, you can enable at-rest encryption for [Secrets](#). This at-rest encryption is additional to any system-level encryption for the etcd cluster or for the filesystem(s) on hosts where you are running the kube-apiserver.

This page shows how to switch from encryption of API data at rest, so that API data are stored unencrypted. You might want to do this to improve performance; usually, though, if it was a good idea to encrypt some data, it's also a good idea to leave them encrypted.

Note:

This task covers encryption for resource data stored using the [Kubernetes API](#). For example, you can encrypt Secret objects, including the key-value data they contain.

If you wanted to manage encryption for data in filesystems that are mounted into containers, you instead need to either:

- use a storage integration that provides encrypted [volumes](#)
- encrypt the data within your own 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. It is recommended to run this tutorial on a cluster with at least two nodes that are not acting as control plane hosts. If you do not already have a cluster, you can create one by using [minikube](#) or you can use one of these Kubernetes playgrounds:
 - [Killercoda](#)
 - [Play with Kubernetes](#)
- This task assumes that you are running the Kubernetes API server as a [static pod](#) on each control plane node.
- Your cluster's control plane **must** use etcd v3.x (major version 3, any minor version).
- To encrypt a custom resource, your cluster must be running Kubernetes v1.26 or newer.
- You should have some API data that are already encrypted.

To check the version, enter `kubectl version`.

Determine whether encryption at rest is already enabled

By default, the API server uses an `identity` provider that stores plain-text representations of resources. **The default identity provider does not provide any confidentiality protection.**

The `kube-apiserver` process accepts an argument `--encryption-provider-config` that specifies a path to a configuration file. The contents of that file, if you specify one, control how Kubernetes API data is encrypted in etcd. If it is not specified, you do not have encryption at rest enabled.

The format of that configuration file is YAML, representing a configuration API kind named [EncryptionConfiguration](#). You can see an example configuration in [Encryption at rest configuration](#).

If `--encryption-provider-config` is set, check which resources (such as `secrets`) are configured for encryption, and what provider is used. Make sure that the preferred provider for that resource type is **not** `identity`; you only set `identity` (*no encryption*) as default when you want to disable encryption at rest. Verify that the first-listed provider for a resource is something **other** than `identity`, which means that any new information written to resources of that type will be encrypted as configured. If you do see `identity` as the first-listed provider for any resource, this means that those resources are being written out to etcd without encryption.

Decrypt all data

This example shows how to stop encrypting the Secret API at rest. If you are encrypting other API kinds, adjust the steps to match.

Locate the encryption configuration file

First, find the API server configuration files. On each control plane node, static Pod manifest for the `kube-apiserver` specifies a command line argument, `--encryption-provider-config`. You are likely to find that this file is mounted into the static Pod using a [hostPath](#) volume mount. Once you locate the volume you can find the file on the node filesystem and inspect it.

Configure the API server to decrypt objects

To disable encryption at rest, place the `identity` provider as the first entry in your encryption configuration file.

For example, if your existing `EncryptionConfiguration` file reads:

```
---
apiVersion: apiserver.config.k8s.io/v1
kind: EncryptionConfiguration
resources:
  - resources:
    - secrets
  providers:
    - aescbc:
      keys:
        # Do not use this (invalid) example key for encryption
        - name: example
          secret: 2KfZgdiq2K0g2YrYpyDYs9mF2LPZhQ==
```

then change it to:

```
---  
apiVersion: apiserver.config.k8s.io/v1  
kind: EncryptionConfiguration  
resources:  
  - resources:  
    - secrets  
  providers:  
    - identity: {} # add this line  
    - aescbc:  
      keys:  
        - name: example  
        secret: 2KfZgdiq2K0g2YrYpyDYs9mF2LPZhQ==
```

and restart the kube-apiserver Pod on this node.

Reconfigure other control plane hosts

If you have multiple API servers in your cluster, you should deploy the changes in turn to each API server.

Make sure that you use the same encryption configuration on each control plane host.

Force decryption

Then run the following command to force decryption of all Secrets:

```
# If you are decrypting a different kind of object, change "secrets" to  
kubectl get secrets --all-namespaces -o json | kubectl replace -f -
```

Once you have replaced **all** existing encrypted resources with backing data that don't use encryption, you can remove the encryption settings from the kube-apiserver .

The command line options to remove are:

- `--encryption-provider-config`
- `--encryption-provider-config-automatic-reload`

Restart the kube-apiserver Pod again to apply the new configuration.

Reconfigure other control plane hosts

If you have multiple API servers in your cluster, you should again deploy the changes in turn to each API server.

Make sure that you use the same encryption configuration on each control plane host.

What's next

- Learn more about the [EncryptionConfiguration configuration API \(v1\)](#).

2.24 - Guaranteed Scheduling For Critical Add-On Pods

Kubernetes core components such as the API server, scheduler, and controller-manager run on a control plane node. However, add-ons must run on a regular cluster node. Some of these add-ons are critical to a fully functional cluster, such as metrics-server, DNS, and UI. A cluster may stop working properly if a critical add-on is evicted (either manually or as a side effect of another operation like upgrade) and becomes pending (for example when the cluster is highly utilized and either there are other pending pods that schedule into the space vacated by the evicted critical add-on pod or the amount of resources available on the node changed for some other reason).

Note that marking a pod as critical is not meant to prevent evictions entirely; it only prevents the pod from becoming permanently unavailable. A static pod marked as critical can't be evicted. However, non-static pods marked as critical are always rescheduled.

Marking pod as critical

To mark a Pod as critical, set priorityClassName for that Pod to `system-cluster-critical` or `system-node-critical`. `system-node-critical` is the highest available priority, even higher than `system-cluster-critical`.

2.25 - IP Masquerade Agent User Guide

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

Before you begin

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

- [Killercoda](#)
- [Play with Kubernetes](#)

To check the version, enter `kubectl version`.

IP Masquerade Agent User Guide

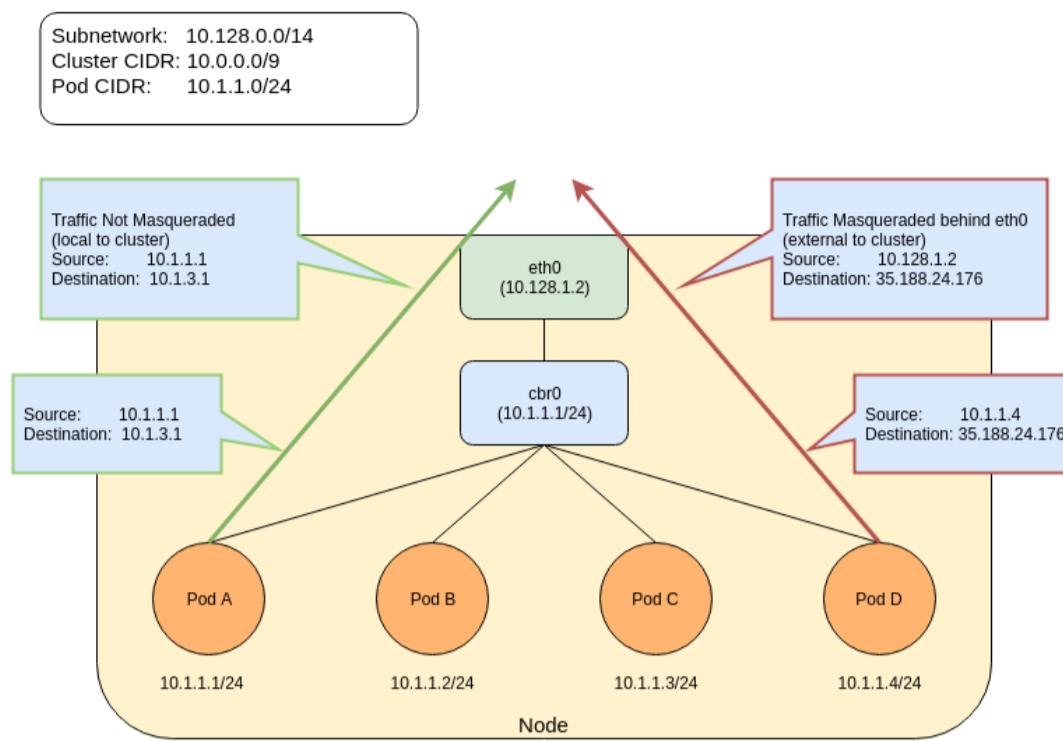
The `ip-masq-agent` configures iptables rules to hide a pod's IP address behind the cluster node's IP address. This is typically done when sending traffic to destinations outside the cluster's pod [CIDR](#) range.

Key Terms

- **NAT (Network Address Translation):** Is a method of remapping one IP address to another by modifying either the source and/or destination address information in the IP header. Typically performed by a device doing IP routing.
- **Masquerading:** A form of NAT that is typically used to perform a many to one address translation, where multiple source IP addresses are masked behind a single address, which is typically the device doing the IP routing. In Kubernetes this is the Node's IP address.
- **CIDR (Classless Inter-Domain Routing):** Based on the variable-length subnet masking, allows specifying arbitrary-length prefixes. CIDR introduced a new method of representation for IP addresses, now commonly known as **CIDR notation**, in which an address or routing prefix is written with a suffix indicating the number of bits of the prefix, such as 192.168.2.0/24.
- **Link Local:** A link-local address is a network address that is valid only for communications within the network segment or the broadcast domain that the host is connected to. Link-local addresses for IPv4 are defined in the address block 169.254.0.0/16 in CIDR notation.

The `ip-masq-agent` configures iptables rules to handle masquerading node/pod IP addresses when sending traffic to destinations outside the cluster node's IP and the Cluster IP range. This essentially hides pod IP addresses behind the cluster node's IP address. In some environments, traffic to "external" addresses must come from a known machine address. For example, in Google Cloud, any traffic to the internet must come from a VM's IP. When containers are used, as in Google Kubernetes Engine, the Pod IP will be rejected for egress. To avoid this, we must hide the Pod IP behind the VM's own IP address - generally known as "masquerade". By default, the agent is configured to treat the three private IP ranges specified by [RFC 1918](#) as non-masquerade [CIDR](#). These ranges are 10.0.0.0/8, 172.16.0.0/12, and 192.168.0.0/16. The agent will also treat

link-local (169.254.0.0/16) as a non-masquerade CIDR by default. The agent is configured to reload its configuration from the location `/etc/config/ip-masq-agent` every 60 seconds, which is also configurable.



The agent configuration file must be written in YAML or JSON syntax, and may contain three optional keys:

- `nonMasqueradeCIDRs` : A list of strings in [CIDR](#) notation that specify the non-masquerade ranges.
- `masqLinkLocal` : A Boolean (true/false) which indicates whether to masquerade traffic to the link local prefix `169.254.0.0/16`. False by default.
- `resyncInterval` : A time interval at which the agent attempts to reload config from disk. For example: '30s', where 's' means seconds, 'ms' means milliseconds.

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
```

target	prot	opt	source	destination
RETURN	all	--	anywhere	169.254.0.0/16 /* ip-masq-agent
RETURN	all	--	anywhere	10.0.0.0/8 /* ip-masq-agent
RETURN	all	--	anywhere	172.16.0.0/12 /* ip-masq-agent
RETURN	all	--	anywhere	192.168.0.0/16 /* ip-masq-agent
MASQUERADE	all	--	anywhere	anywhere /* ip-masq-agent

By default, in GCE/Google Kubernetes Engine, if network policy is enabled or you are using a cluster CIDR not in the 10.0.0.0/8 range, the `ip-masq-agent` will run in your cluster. If you are running in another environment, you can add the `ip-masq-agent` [DaemonSet](#) to your cluster.

Create an ip-masq-agent

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

```
kubectl apply -f https://raw.githubusercontent.com/kubernetes-sigs/ip-
```

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 node.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 configmap to your cluster:

```
kubectl create configmap ip-masq-agent --from-file=config --namespace=
```

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  
RETURN    all  --  anywhere        10.0.0.0/8       /* ip-masq-agent  
MASQUERADE all  --  anywhere        anywhere         /* ip-masq-agent
```

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 ConfigMap.

```
nonMasqueradeCIDRs:
```

```
- 10.0.0.0/8
```

```
resyncInterval: 60s
```

```
masqLinkLocal: true
```

2.26 - Limit Storage Consumption

This example demonstrates how to limit the amount of storage consumed in a namespace.

The following resources are used in the demonstration: [ResourceQuota](#), [LimitRange](#), and [PersistentVolumeClaim](#).

Before you begin

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

2.27 - Migrate Replicated Control Plane To Use Cloud Controller Manager

The cloud-controller-manager is a Kubernetes [control plane](#) component that embeds cloud-specific control logic. The [cloud controller manager](#) lets you link your cluster into your cloud provider's API, and separates out the components that interact with that cloud platform from components that only interact with your cluster.

By decoupling the interoperability logic between Kubernetes and the underlying cloud infrastructure, the cloud-controller-manager component enables cloud providers to release features at a different pace compared to the main Kubernetes project.

Background

As part of the [cloud provider extraction effort](#), all cloud specific controllers must be moved out of the `kube-controller-manager`. All existing clusters that run cloud controllers in the `kube-controller-manager` must migrate to instead run the controllers in a cloud provider specific `cloud-controller-manager`.

Leader Migration provides a mechanism in which HA clusters can safely migrate "cloud specific" controllers between the `kube-controller-manager` and the `cloud-controller-manager` via a shared resource lock between the two components while upgrading the replicated control plane. For a single-node control plane, or if unavailability of controller managers can be tolerated during the upgrade, Leader Migration is not needed and this guide can be ignored.

Leader Migration can be enabled by setting `--enable-leader-migration` on `kube-controller-manager` or `cloud-controller-manager`. Leader Migration only applies during the upgrade and can be safely disabled or left enabled after the upgrade is complete.

This guide walks you through the manual process of upgrading the control plane from `kube-controller-manager` with built-in cloud provider to running both `kube-controller-manager` and `cloud-controller-manager`. If you use a tool to deploy and manage the cluster, please refer to the documentation of the tool and the cloud provider for specific instructions of the migration.

Before you begin

It is assumed that the control plane is running Kubernetes version N and to be upgraded to version N + 1. Although it is possible to migrate within the same version, ideally the migration should be performed as part of an upgrade so that changes of configuration can be aligned to each release. The exact versions of N and N + 1 depend on each cloud provider. For example, if a cloud provider builds a `cloud-controller-manager` to work with Kubernetes 1.24, then N can be 1.23 and N + 1 can be 1.24.

The control plane nodes should run `kube-controller-manager` with Leader Election enabled, which is the default. As of version N, an in-tree cloud provider must be set with `--cloud-provider` flag and `cloud-controller-manager` should not yet be deployed.

The out-of-tree cloud provider must have built a `cloud-controller-manager` with Leader Migration implementation. If the cloud provider imports `k8s.io/cloud-provider` and `k8s.io/controller-manager` of version v0.21.0 or

later, Leader Migration will be available. However, for version before v0.22.0, Leader Migration is alpha and requires feature gate `ControllerManagerLeaderMigration` to be enabled in `cloud-controller-manager`.

This guide assumes that kubelet of each control plane node starts `kube-controller-manager` and `cloud-controller-manager` as static pods defined by their manifests. If the components run in a different setting, please adjust the steps accordingly.

For authorization, this guide assumes that the cluster uses RBAC. If another authorization mode grants permissions to `kube-controller-manager` and `cloud-controller-manager` components, please grant the needed access in a way that matches the mode.

Grant access to Migration Lease

The default permissions of the controller manager allow only accesses to their main Lease. In order for the migration to work, accesses to another Lease are required.

You can grant `kube-controller-manager` full access to the leases API by modifying the `system::leader-locking-kube-controller-manager` role. This task guide assumes that the name of the migration lease is `cloud-provider-extraction-migration`.

```
kubectl patch -n kube-system role 'system::leader-locking-kube-control
```

Do the same to the `system::leader-locking-cloud-controller-manager` role.

```
kubectl patch -n kube-system role 'system::leader-locking-cloud-contro
```

Initial Leader Migration configuration

Leader Migration optionally takes a configuration file representing the state of controller-to-manager assignment. At this moment, with in-tree cloud provider, `kube-controller-manager` runs `route`, `service`, and `cloud-node-lifecycle`. The following example configuration shows the assignment.

Leader Migration can be enabled without a configuration. Please see [Default Configuration](#) for details.

```
kind: LeaderMigrationConfiguration
apiVersion: controllermanager.config.k8s.io/v1
leaderName: cloud-provider-extraction-migration
controllerLeaders:
  - name: route
    component: kube-controller-manager
  - name: service
    component: kube-controller-manager
  - name: cloud-node-lifecycle
    component: kube-controller-manager
```

Alternatively, because the controllers can run under either controller managers, setting `component` to `*` for both sides makes the configuration file consistent between both parties of the migration.

```
# wildcard version
kind: LeaderMigrationConfiguration
apiVersion: controllermanager.config.k8s.io/v1
leaderName: cloud-provider-extraction-migration
controllerLeaders:
- name: route
  component: *
- name: service
  component: *
- name: cloud-node-lifecycle
  component: *
```

On each control plane node, save the content to `/etc/leadermigration.conf`, and update the manifest of `kube-controller-manager` so that the file is mounted inside the container at the same location. Also, update the same manifest to add the following arguments:

- `--enable-leader-migration` to enable Leader Migration on the controller manager
- `--leader-migration-config=/etc/leadermigration.conf` to set configuration file

Restart `kube-controller-manager` on each node. At this moment, `kube-controller-manager` has leader migration enabled and is ready for the migration.

Deploy Cloud Controller Manager

In version $N + 1$, the desired state of controller-to-manager assignment can be represented by a new configuration file, shown as follows. Please note `component` field of each `controllerLeaders` changing from `kube-controller-manager` to `cloud-controller-manager`. Alternatively, use the wildcard version mentioned above, which has the same effect.

```
kind: LeaderMigrationConfiguration
apiVersion: controllermanager.config.k8s.io/v1
leaderName: cloud-provider-extraction-migration
controllerLeaders:
- name: route
  component: cloud-controller-manager
- name: service
  component: cloud-controller-manager
- name: cloud-node-lifecycle
  component: cloud-controller-manager
```

When creating control plane nodes of version $N + 1$, the content should be deployed to `/etc/leadermigration.conf`. The manifest of `cloud-controller-manager` should be updated to mount the configuration file in the same manner as `kube-controller-manager` of version N . Similarly, add `--enable-leader-migration` and `--leader-migration-config=/etc/leadermigration.conf` to the arguments of `cloud-controller-manager`.

Create a new control plane node of version $N + 1$ with the updated `cloud-controller-manager` manifest, and with the `--cloud-provider` flag set to `external` for `kube-controller-manager`. `kube-controller-manager` of version

N + 1 MUST NOT have Leader Migration enabled because, with an external cloud provider, it does not run the migrated controllers anymore, and thus it is not involved in the migration.

Please refer to [Cloud Controller Manager Administration](#) for more detail on how to deploy `cloud-controller-manager`.

Upgrade Control Plane

The control plane now contains nodes of both version N and N + 1. The nodes of version N run `kube-controller-manager` only, and these of version N + 1 run both `kube-controller-manager` and `cloud-controller-manager`. The migrated controllers, as specified in the configuration, are running under either `kube-controller-manager` of version N or `cloud-controller-manager` of version N + 1 depending on which controller manager holds the migration lease. No controller will ever be running under both controller managers at any time.

In a rolling manner, create a new control plane node of version N + 1 and bring down one of version N until the control plane contains only nodes of version N + 1. If a rollback from version N + 1 to N is required, add nodes of version N with Leader Migration enabled for `kube-controller-manager` back to the control plane, replacing one of version N + 1 each time until there are only nodes of version N.

(Optional) Disable Leader Migration

Now that the control plane has been upgraded to run both `kube-controller-manager` and `cloud-controller-manager` of version N + 1, Leader Migration has finished its job and can be safely disabled to save one Lease resource. It is safe to re-enable Leader Migration for the rollback in the future.

In a rolling manager, update manifest of `cloud-controller-manager` to unset both `--enable-leader-migration` and `--leader-migration-config=` flag, also remove the mount of `/etc/leadermigration.conf`, and finally remove `/etc/leadermigration.conf`. To re-enable Leader Migration, recreate the configuration file and add its mount and the flags that enable Leader Migration back to `cloud-controller-manager`.

Default Configuration

Starting Kubernetes 1.22, Leader Migration provides a default configuration suitable for the default controller-to-manager assignment. The default configuration can be enabled by setting `--enable-leader-migration` but without `--leader-migration-config=`.

For `kube-controller-manager` and `cloud-controller-manager`, if there are no flags that enable any in-tree cloud provider or change ownership of controllers, the default configuration can be used to avoid manual creation of the configuration file.

Special case: migrating the Node IPAM controller

If your cloud provider provides an implementation of Node IPAM controller, you should switch to the implementation in `cloud-controller-manager`. Disable Node IPAM controller in `kube-controller-manager` of version N + 1 by adding `--controllers=*,-nodeipam` to its flags. Then add `nodeipam` to the list of migrated controllers.

```
# wildcard version, with nodeipam
kind: LeaderMigrationConfiguration
apiVersion: controllermanager.config.k8s.io/v1
leaderName: cloud-provider-extraction-migration
controllerLeaders:
  - name: route
    component: *
  - name: service
    component: *
  - name: cloud-node-lifecycle
    component: *
  - name: nodeipam
    component: *
```

What's next

- Read the [Controller Manager Leader Migration](#) enhancement proposal.

2.28 - Namespaces Walkthrough

Kubernetes namespaces help different projects, teams, or customers to share a Kubernetes cluster.

It does this by providing the following:

1. A scope for [Names](#).
2. A mechanism to attach authorization and policy to a subsection of the cluster.

Use of multiple namespaces is optional.

This example demonstrates how to use Kubernetes namespaces to subdivide your cluster.

Before you begin

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

- [Killercoda](#)
- [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.yaml](#) which describes a `development` namespace:

```
apiVersion: v1
kind: Namespace
metadata:
  name: development
  labels:
    name: development
```

Create the `development` namespace using kubectl.

```
kubectl create -f https://k8s.io/examples/admin/namespace-dev.yaml
```

Save the following contents into file [namespace-prod.yaml](#) which describes a `production` namespace:

```
apiVersion: v1
kind: Namespace
metadata:
  name: production
  labels:
    name: production
```

And then let's create the `production` namespace using kubectl.

```
kubectl create -f https://k8s.io/examples/admin/namespace-prod.yaml
```

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: h5M0FtUUIf1BSdI7
    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 add 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.

[admin/snowflake-deployment.yaml](#)

```
apiVersion: apps/v1
kind: Deployment
metadata:
  labels:
    app: snowflake
  name: snowflake
spec:
  replicas: 2
  selector:
    matchLabels:
      app: snowflake
  template:
    metadata:
      labels:
        app: snowflake
    spec:
      containers:
        - image: registry.k8s.io/serve_hostname
          imagePullPolicy: Always
          name: snowflake
```

Apply the manifest to create a Deployment

```
kubectl apply -f https://k8s.io/examples/admin/snowflake-deployment.yaml
```

We have created a deployment whose replica size is 2 that is running the pod called `snowflake` with a basic container that serves the hostname.

```
kubectl get deployment
```

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

```
kubectl get pods -l app=snowflake
```

NAME	READY	STATUS	RESTARTS	AGE
snowflake-3968820950-9dgr8	1/1	Running	0	2m
snowflake-3968820950-vgc4n	1/1	Running	0	2m

And this is great, developers are able to do what they want, and they do not have to worry about affecting content in the `production` namespace.

Let's switch to the `production` namespace and show how resources in one namespace are hidden from the other.

```
kubectl config use-context prod
```

The `production` namespace should be empty, and the following commands should return nothing.

```
kubectl get deployment  
kubectl get pods
```

Production likes to run cattle, so let's create some cattle pods.

```
kubectl create deployment cattle --image=registry.k8s.io/serve_hostname  
kubectl get deployment
```

NAME	READY	UP-TO-DATE	AVAILABLE	AGE
cattle	5/5	5	5	10s

```
kubectl get pods -l app=cattle
```

NAME	READY	STATUS	RESTARTS	AGE
cattle-2263376956-41xy6	1/1	Running	0	34s
cattle-2263376956-kw466	1/1	Running	0	34s
cattle-2263376956-n4v97	1/1	Running	0	34s
cattle-2263376956-p5p3i	1/1	Running	0	34s
cattle-2263376956-sxpth	1/1	Running	0	34s

At this point, it should be clear that the resources users create in one namespace are hidden from the other namespace.

As the policy support in Kubernetes evolves, we will extend this scenario to show how you can provide different authorization rules for each namespace.

2.29 - Operating etcd clusters for Kubernetes

etcd is a consistent and highly-available key value store used as Kubernetes' backing store for all cluster data.

If your Kubernetes cluster uses etcd as its backing store, make sure you have a [back up](#) plan for the data.

You can find in-depth information about etcd in the official [documentation](#).

Before you begin

You need to have a Kubernetes cluster, and the kubectl command-line tool must be configured to communicate with your cluster. It is recommended to run this task on a cluster with at least two nodes that are not acting as control plane nodes . If you do not already have a cluster, you can create one by using [minikube](#).

Prerequisites

- Run etcd as a cluster of odd members.
- etcd is a leader-based distributed system. Ensure that the leader periodically send heartbeats on time to all followers to keep the cluster stable.
- Ensure that no resource starvation occurs.

Performance and stability of the cluster is sensitive to network and disk I/O. Any resource starvation can lead to heartbeat timeout, causing instability of the cluster. An unstable etcd indicates that no leader is elected. Under such circumstances, a cluster cannot make any changes to its current state, which implies no new pods can be scheduled.

- Keeping etcd clusters stable is critical to the stability of Kubernetes clusters. Therefore, run etcd clusters on dedicated machines or isolated environments for [guaranteed resource requirements](#).
- The minimum recommended etcd versions to run in production are 3.4.22+ and 3.5.6+ .

Resource requirements

Operating etcd with limited resources is suitable only for testing purposes. For deploying in production, advanced hardware configuration is required. Before deploying etcd in production, see [resource requirement reference](#).

Starting etcd clusters

This section covers starting a single-node and multi-node etcd cluster.

Single-node etcd cluster

Use a single-node etcd cluster only for testing purpose.

1. Run the following:

```
etcd --listen-client-urls=http://$PRIVATE_IP:2379 \
--advertise-client-urls=http://$PRIVATE_IP:2379
```

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

Make sure `PRIVATE_IP` is set to your etcd client IP.

Multi-node etcd cluster

For durability and high availability, run etcd as a multi-node cluster in production and back it up periodically. A five-member cluster is recommended in production. For more information, see [FAQ documentation](#).

Configure an etcd cluster either by static member information or by dynamic discovery. For more information on clustering, see [etcd clustering documentation](#).

For an example, consider a five-member etcd cluster running with the following client URLs: `http://$IP1:2379`, `http://$IP2:2379`, `http://$IP3:2379`, `http://$IP4:2379`, and `http://$IP5:2379`. To start a Kubernetes API server:

1. Run the following:

```
etcd --listen-client-urls=http://$IP1:2379,http://$IP2:2379,http:
```

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

Make sure the `IP<n>` variables are set to your client IP addresses.

Multi-node etcd cluster with load balancer

To run a load balancing etcd cluster:

1. Set up an etcd cluster.
2. Configure a load balancer in front of the etcd cluster. For example, let the address of the load balancer be `$LB`.
3. Start Kubernetes API Servers with the flag `--etcd-servers=$LB:2379`.

Securing etcd clusters

Access to etcd is equivalent to root permission in the cluster so ideally only the API server should have access to it. Considering the sensitivity of the data, it is recommended to grant permission to only those nodes that require access to etcd clusters.

To secure etcd, either set up firewall rules or use the security features provided by etcd. etcd security features depend on x509 Public Key Infrastructure (PKI). To begin, establish secure communication channels by generating a key and certificate pair. For example, use key pairs `peer.key` and `peer.cert` for securing communication between etcd members, and `client.key` and `client.cert` for securing communication between etcd and its clients. See the [example scripts](#) provided by the etcd project to generate key pairs and CA files for client authentication.

Securing communication

To configure etcd with secure peer communication, specify flags `--peer-key-file=peer.key` and `--peer-cert-file=peer.cert`, and use HTTPS as the URL schema.

Similarly, to configure etcd with secure client communication, specify flags `--key-file=k8sclient.key` and `--cert-file=k8sclient.cert`, and use HTTPS as the URL schema. Here is an example on a client command that uses secure communication:

```
ETCDCTL_API=3 etcdctl --endpoints 10.2.0.9:2379 \
--cert=/etc/kubernetes/pki/etcd/server.crt \
--key=/etc/kubernetes/pki/etcd/server.key \
--cacert=/etc/kubernetes/pki/etcd/ca.crt \
member list
```

Limiting access of etcd clusters

After configuring secure communication, restrict the access of etcd cluster to only the Kubernetes API servers. Use TLS authentication to do so.

For example, consider key pairs `k8sclient.key` and `k8sclient.cert` that are trusted by the CA `etcd.ca`. When etcd is configured with `--client-cert-auth` along with TLS, it verifies the certificates from clients by using system CAs or the CA passed in by `--trusted-ca-file` flag. Specifying flags `--client-cert-auth=true` and `--trusted-ca-file=etcd.ca` will restrict the access to clients with the certificate `k8sclient.cert`.

Once etcd is configured correctly, only clients with valid certificates can access it. To give Kubernetes API servers the access, configure them with the flags `--etcd-certfile=k8sclient.cert`, `--etcd-keyfile=k8sclient.key` and `--etcd-cafile=ca.cert`.

Note: etcd authentication is not currently supported by Kubernetes. For more information, see the related issue [Support Basic Auth for Etcd v2](#).

Replacing a failed etcd member

etcd cluster achieves high availability by tolerating minor member failures. However, to improve the overall health of the cluster, replace failed members immediately. When multiple members fail, replace them one by one. Replacing a failed member involves two steps: removing the failed member and adding a new member.

Though etcd keeps unique member IDs internally, it is recommended to use a unique name for each member to avoid human errors. For example, consider a three-member etcd cluster. Let the URLs be,
`member1=http://10.0.0.1`, `member2=http://10.0.0.2`, and
`member3=http://10.0.0.3`. When `member1` fails, replace it with
`member4=http://10.0.0.4`.

1. Get the member ID of the failed `member1`:

```
etcdctl --endpoints=http://10.0.0.2,http://10.0.0.3 member list
```

The following message is displayed:

```
8211f1d0f64f3269, started, member1, http://10.0.0.1:2380, http://  
91bc3c398fb3c146, started, member2, http://10.0.0.2:2380, http://  
fd422379fda50e48, started, member3, http://10.0.0.3:2380, http://
```

2. Do either of the following:

1. If each Kubernetes API server is configured to communicate with all etcd members, remove the failed member from the `--etcd-servers` flag, then restart each Kubernetes API server.
2. If each Kubernetes API server communicates with a single etcd member, then stop the Kubernetes API server that communicates with the failed etcd.
3. Stop the etcd server on the broken node. It is possible that other clients besides the Kubernetes API server is causing traffic to etcd and it is desirable to stop all traffic to prevent writes to the data dir.
4. Remove the failed member:

```
etcdctl member remove 8211f1d0f64f3269
```

The following message is displayed:

```
Removed member 8211f1d0f64f3269 from cluster
```

5. 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
```

6. 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  
export ETCD_INITIAL_CLUSTER_STATE=existing  
etcd [flags]
```

7. Do either of the following:

1. If each Kubernetes API server is configured to communicate with all etcd members, add the newly added member to the `--etcd-servers` flag, then restart each Kubernetes API server.
2. If each Kubernetes API server communicates with a single etcd member, start the Kubernetes API server that was stopped in step 2. Then configure Kubernetes API server clients to again route requests to the Kubernetes API server that was stopped. This can often be done by configuring a load balancer.

For more information on cluster reconfiguration, see [etcd reconfiguration documentation](#).

Backing up an etcd cluster

All Kubernetes objects are stored on etcd. Periodically backing up the etcd cluster data is important to recover Kubernetes clusters under disaster scenarios, such as losing all control plane nodes. The snapshot file contains all the Kubernetes states and critical information. In order to keep the sensitive Kubernetes data safe, encrypt the snapshot files.

Backing up an etcd cluster can be accomplished in two ways: etcd built-in snapshot and volume snapshot.

Built-in snapshot

etcd supports built-in snapshot. A snapshot may either be taken from a live member with the `etcdctl snapshot save` command or by copying the `member/snap/db` file from an etcd [data directory](#) that is not currently used by an etcd process. Taking the snapshot will not affect the performance of the member.

Below is an example for taking a snapshot of the keyspace served by `$ENDPOINT` to the file `snapshot.db`:

```
ETCDCTL_API=3 etcdctl --endpoints $ENDPOINT snapshot save snapshot.db
```

Verify the snapshot:

```
ETCDCTL_API=3 etcdctl --write-out=table snapshot status snapshot.db
```

HASH	REVISION	TOTAL KEYS	TOTAL SIZE
fe01cf57	10	7	2.1 MB

Volume snapshot

If etcd is running on a storage volume that supports backup, such as Amazon Elastic Block Store, back up etcd data by taking a snapshot of the storage volume.

Snapshot using etcdctl options

We can also take the snapshot using various options given by etcdctl. For example

```
ETCDCTL_API=3 etcdctl -h
```

will list various options available from etcdctl. For example, you can take a snapshot by specifying the endpoint, certificates etc as shown below:

```
ETCDCTL_API=3 etcdctl --endpoints=https://127.0.0.1:2379 \
--cacert=<trusted-ca-file> --cert=<cert-file> --key=<key-file> \
snapshot save <backup-file-location>
```

where `trusted-ca-file`, `cert-file` and `key-file` can be obtained from the description of the etcd Pod.

Scaling out etcd clusters

Scaling out etcd clusters increases availability by trading off performance. Scaling does not increase cluster performance nor capability. A general rule is not to scale out or in etcd clusters. Do not configure any auto scaling groups for etcd clusters. It is highly recommended to always run a static five-member etcd cluster for production Kubernetes clusters at any officially supported scale.

A reasonable scaling is to upgrade a three-member cluster to a five-member one, when more reliability is desired. See [etcd reconfiguration documentation](#) for information on how to add members into an existing cluster.

Restoring an etcd cluster

etcd supports restoring from snapshots that are taken from an etcd process of the [major.minor](#) version. Restoring a version from a different patch version of etcd also is supported. A restore operation is employed to recover the data of a failed cluster.

Before starting the restore operation, a snapshot file must be present. It can either be a snapshot file from a previous backup operation, or from a remaining [data directory](#).

Here is an example:

```
ETCDCTL_API=3 etcdctl --endpoints 10.2.0.9:2379 snapshot restore snap1
```

Another example for restoring using `etcdctl` options:

```
ETCDCTL_API=3 etcdctl --data-dir <data-dir-location> snapshot restore
```

where `<data-dir-location>` is a directory that will be created during the restore process.

Yet another example would be to first export the `ETCDCTL_API` environment variable:

```
export ETCDCTL_API=3
etcdctl --data-dir <data-dir-location> snapshot restore snapshot.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 servers with the flag `--etcd-servers=$NEW_ETCD_CLUSTER` instead of the flag `--etcd-servers=$OLD_ETCD_CLUSTER`. Replace `$NEW_ETCD_CLUSTER` and `$OLD_ETCD_CLUSTER` with the respective IP addresses. If a load balancer is used in front of an etcd cluster, you might need to update the load balancer instead.

If the majority of etcd members have permanently failed, the etcd cluster is considered failed. In this scenario, Kubernetes cannot make any changes to its current state. Although the scheduled pods might continue to run, no new pods can be scheduled. In such cases, recover the etcd cluster and potentially reconfigure Kubernetes API servers to fix the issue.

Note:

If any API servers are running in your cluster, you should not attempt to restore instances of etcd. Instead, follow these steps to restore etcd:

- stop *all* API server instances
- restore state in all etcd instances
- restart all API server instances

We also recommend restarting any components (e.g. `kube-scheduler`, `kube-controller-manager`, `kubelet`) to ensure that they don't rely on some stale data. Note that in practice, the restore takes a bit of time. During the restoration, critical components will lose leader lock and restart themselves.

Upgrading etcd clusters

For more details on etcd upgrade, please refer to the [etcd upgrades](#) documentation.

Note: Before you start an upgrade, please back up your etcd cluster first.

Maintaining etcd clusters

For more details on etcd maintenance, please refer to the [etcd maintenance](#) documentation.

ⓘ This item links to a third party project or product that is not part of Kubernetes itself. [More information](#)

Note:

Defragmentation is an expensive operation, so it should be executed as infrequent as possible. On the other hand, it's also necessary to make sure any etcd member will not run out of the storage quota. The Kubernetes project recommends that when you perform defragmentation, you use a tool such as [etcd-defrag](#).

You can also run the defragmentation tool as a Kubernetes CronJob, to make sure that defragmentation happens regularly. See [etcd-defrag-cronjob.yaml](#) for details.

2.30 - Reserve Compute Resources for System Daemons

Kubernetes nodes can be scheduled to `capacity`. Pods can consume all the available capacity on a node by default. This is an issue because nodes typically run quite a few system daemons that power the OS and Kubernetes itself. Unless resources are set aside for these system daemons, pods and system daemons compete for resources and lead to resource starvation issues on the node.

The `kubelet` exposes a feature named 'Node Allocatable' that helps to reserve compute resources for system daemons. Kubernetes recommends cluster administrators to configure 'Node Allocatable' based on their workload density on each node.

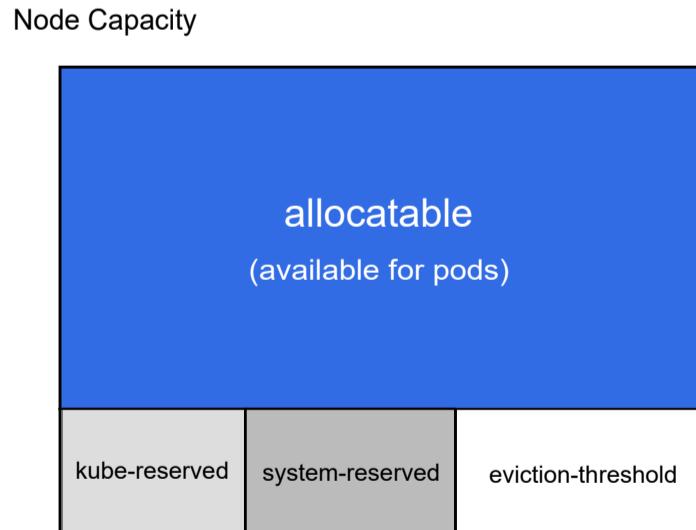
Before you begin

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

- [KillerCoda](#)
- [Play with Kubernetes](#)

Your Kubernetes server must be at or later than version 1.8. To check the version, enter `kubectl version`. Your Kubernetes server must be at or later than version 1.17 to use the `kubelet` command line option `--reserved-cpus` to set an [explicitly reserved CPU list](#).

Node Allocatable



'Allocatable' on a Kubernetes node is defined as the amount of compute resources that are available for pods. The scheduler does not over-subscribe 'Allocatable'. 'CPU', 'memory' and 'ephemeral-storage' are supported as of now.

Node Allocatable is exposed as part of `v1.Node` object in the API and as part of `kubectl describe node` in the CLI.

Resources can be reserved for two categories of system daemons in the `kubelet`.

Enabling QoS and Pod level cgroups

To properly enforce node allocatable constraints on the node, you must enable the new cgroup hierarchy via the `--cgroups-per-qos` flag. This flag is enabled by default. When enabled, the `kubelet` will parent all end-user pods under a cgroup hierarchy managed by the `kubelet`.

Configuring a cgroup driver

The `kubelet` supports manipulation of the cgroup hierarchy on the host using a cgroup driver. The driver is configured via the `--cgroup-driver` flag.

The supported values are the following:

- `cgroupfs` is the default driver that performs direct manipulation of the cgroup filesystem on the host in order to manage cgroup sandboxes.
- `systemd` is an alternative driver that manages cgroup sandboxes using transient slices for resources that are supported by that init system.

Depending on the configuration of the associated container runtime, operators may have to choose a particular cgroup driver to ensure proper system behavior. For example, if operators use the `systemd` cgroup driver provided by the `containerd` runtime, the `kubelet` must be configured to use the `systemd` cgroup driver.

Kube Reserved

- **Kubelet Flag:** `--kube-reserved=[cpu=100m][,][memory=100Mi][,][ephemeral-storage=1Gi][,][pid=1000]`
- **Kubelet Flag:** `--kube-reserved-cgroup=`

`kube-reserved` is meant to capture resource reservation for kubernetes system daemons like the `kubelet`, `container runtime`, `node problem detector`, etc. It is not meant to reserve resources for system daemons that are run as pods. `kube-reserved` is typically a function of `pod density` on the nodes.

In addition to `cpu`, `memory`, and `ephemeral-storage`, `pid` may be specified to reserve the specified number of process IDs for kubernetes system daemons.

To optionally enforce `kube-reserved` on kubernetes system daemons, specify the parent control group for kube daemons as the value for `--kube-reserved-cgroup` `kubelet` flag.

It is recommended that the kubernetes system daemons are placed under a top level control group (`runtime.slice` on systemd machines for example). Each system daemon should ideally run within its own child control group. Refer to [the design proposal](#) for more details on recommended control group hierarchy.

Note that Kubelet **does not** create `--kube-reserved-cgroup` if it doesn't exist. The `kubelet` will fail to start if an invalid cgroup is specified. With `systemd` cgroup driver, you should follow a specific pattern for the name of the cgroup you define: the name should be the value you set for `--kube-reserved-cgroup`, with `.slice` appended.

System Reserved

- **Kubelet Flag:** `--system-reserved=[cpu=100m][,][memory=100Mi][,][ephemeral-storage=1Gi][,][pid=1000]`
- **Kubelet Flag:** `--system-reserved-cgroup=`

`system-reserved` is meant to capture resource reservation for OS system daemons like `sshd`, `udev`, etc. `system-reserved` should reserve memory for the kernel too since kernel memory is not accounted to pods in Kubernetes at this time. Reserving resources for user login sessions is also recommended (`user.slice` in `systemd` world).

In addition to `cpu`, `memory`, and `ephemeral-storage`, `pid` may be specified to reserve the specified number of process IDs for OS system daemons.

To optionally enforce `system-reserved` on system daemons, specify the parent control group for OS system daemons as the value for `--system-reserved-cgroup` kubelet flag.

It is recommended that the OS system daemons are placed under a top level control group (`system.slice` on `systemd` machines for example).

Note that `kubelet` **does not** create `--system-reserved-cgroup` if it doesn't exist. `kubelet` will fail if an invalid cgroup is specified. With `systemd` cgroup driver, you should follow a specific pattern for the name of the cgroup you define: the name should be the value you set for `--system-reserved-cgroup`, with `.slice` appended.

Explicitly Reserved CPU List

FEATURE STATE: `Kubernetes v1.17 [stable]`

Kubelet Flag: `--reserved-cpus=0-3` **KubeletConfiguration Flag:**

`reservedSystemCPUs: 0-3`

`reserved-cpus` is meant to define an explicit CPU set for OS system daemons and Kubernetes system daemons. `reserved-cpus` is for systems that do not intend to define separate top level cgroups for OS system daemons and Kubernetes system daemons with regard to cpuset resource. If the Kubelet **does not** have `--system-reserved-cgroup` and `--kube-reserved-cgroup`, the explicit cpuset provided by `reserved-cpus` will take precedence over the CPUs defined by `--kube-reserved` and `--system-reserved` options.

This option is specifically designed for Telco/NFV use cases where uncontrolled interrupts/timers may impact the workload performance. You can use this option to define the explicit cpuset for the system/Kubernetes daemons as well as the interrupts/timers, so the rest CPUs on the system can be used exclusively for workloads, with less impact from uncontrolled interrupts/timers. To move the system daemon, Kubernetes daemons and interrupts/timers to the explicit cpuset defined by this option, other mechanism outside Kubernetes should be used. For example: in CentOS, you can do this using the tuned toolset.

Eviction Thresholds

Kubelet Flag: `--eviction-hard=[memory.available<500Mi]`

Memory pressure at the node level leads to System OOMs which affects the entire node and all pods running on it. Nodes can go offline temporarily until memory has been reclaimed. To avoid (or reduce the probability of) system OOMs kubelet provides [out of resource](#) management. Evictions are supported for `memory` and `ephemeral-storage` only. By reserving some memory via `--eviction-hard` flag, the kubelet attempts to evict pods whenever memory availability on the node drops

below the reserved value. Hypothetically, if system daemons did not exist on a node, pods cannot use more than `capacity - eviction-hard`. For this reason, resources reserved for evictions are not available for pods.

Enforcing Node Allocatable

Kubelet Flag: `--enforce-node-allocatable=pods[,][system-reserved][,][kube-reserved]`

The scheduler treats 'Allocatable' as the available `capacity` for pods.

`kubelet` enforces 'Allocatable' across pods by default. Enforcement is performed by evicting pods whenever the overall usage across all pods exceeds 'Allocatable'. More details on eviction policy can be found on the [node pressure eviction](#) page. This enforcement is controlled by specifying `pods` value to the `kubelet` flag `--enforce-node-allocatable`.

Optionally, `kubelet` can be made to enforce `kube-reserved` and `system-reserved` by specifying `kube-reserved` & `system-reserved` values in the same flag. Note that to enforce `kube-reserved` or `system-reserved`, `--kube-reserved-cgroup` or `--system-reserved-cgroup` needs to be specified respectively.

General Guidelines

System daemons are expected to be treated similar to [Guaranteed pods](#). System daemons can burst within their bounding control groups and this behavior needs to be managed as part of Kubernetes deployments. For example, `kubelet` should have its own control group and share `kube-reserved` resources with the container runtime. However, Kubelet cannot burst and use up all available Node resources if `kube-reserved` is enforced.

Be extra careful while enforcing `system-reserved` reservation since it can lead to critical system services being CPU starved, OOM killed, or unable to fork on the node. The recommendation is to enforce `system-reserved` only if a user has profiled their nodes exhaustively to come up with precise estimates and is confident in their ability to recover if any process in that group is oom-killed.

- To begin with enforce 'Allocatable' on `pods`.
- Once adequate monitoring and alerting is in place to track `kube-system` daemons, attempt to enforce `kube-reserved` based on usage heuristics.
- If absolutely necessary, enforce `system-reserved` over time.

The resource requirements of `kube` system daemons may grow over time as more and more features are added. Over time, Kubernetes project will attempt to bring down utilization of node system daemons, but that is not a priority as of now. So expect a drop in `Allocatable` capacity in future releases.

Example Scenario

Here is an example to illustrate Node Allocatable computation:

- Node has `32Gi` of memory, `16 CPUs` and `100Gi` of Storage
- `--kube-reserved` is set to `cpu=1,memory=2Gi,ephemeral-storage=1Gi`
- `--system-reserved` is set to `cpu=500m,memory=1Gi,ephemeral-storage=1Gi`
- `--eviction-hard` is set to
`memory.available<500Mi,nodefs.available<10%`

Under this scenario, 'Allocatable' will be 14.5 CPUs, 28.5Gi of memory and 88Gi of local storage. Scheduler ensures that the total memory requests across all pods on this node does not exceed 28.5Gi and storage doesn't exceed 88Gi. Kubelet evicts pods whenever the overall memory usage across pods exceeds 28.5Gi, or if overall disk usage exceeds 88Gi. If all processes on the node consume as much CPU as they can, pods together cannot consume more than 14.5 CPUs.

If kube-reserved and/or system-reserved is not enforced and system daemons exceed their reservation, kubelet evicts pods whenever the overall node memory usage is higher than 31.5Gi or storage is greater than 90Gi.

2.31 - Running Kubernetes Node Components as a Non-root User

FEATURE STATE: [Kubernetes v1.22 \[alpha\]](#)

This document describes how to run Kubernetes Node components such as kubelet, CRI, OCI, and CNI without root privileges, by using a user namespace.

This technique is also known as *rootless mode*.

Note:

This document describes how to run Kubernetes Node components (and hence pods) as a non-root user.

If you are just looking for how to run a pod as a non-root user, see [SecurityContext](#).

Before you begin

Your Kubernetes server must be at or later than version 1.22. To check the version, enter `kubectl version`.

- [Enable Cgroup v2](#)
- [Enable systemd with user session](#)
- [Configure several sysctl values, depending on host Linux distribution](#)
- [Ensure that your unprivileged user is listed in /etc/subuid and /etc/subgid](#)
- Enable the `KubeletInUserNamespace` [feature gate](#)

Running Kubernetes inside Rootless Docker/Podman

kind

[kind](#) supports running Kubernetes inside Rootless Docker or Rootless Podman.

See [Running kind with Rootless Docker](#).

minikube

[minikube](#) also supports running Kubernetes inside Rootless Docker or Rootless Podman.

See the Minikube documentation:

- [Rootless Docker](#)
- [Rootless Podman](#)

Running Kubernetes inside Unprivileged Containers

Note: This section links to third party projects that provide functionality required by Kubernetes. The Kubernetes project authors aren't responsible for these projects, which are listed alphabetically. To add a project to this list, read the [content guide](#) before submitting a change. [More information.](#)

sysbox

[Sysbox](#) is an open-source container runtime (similar to "runc") that supports running system-level workloads such as Docker and Kubernetes inside unprivileged containers isolated with the Linux user namespace.

See [Sysbox Quick Start Guide: Kubernetes-in-Docker](#) for more info.

Sysbox supports running Kubernetes inside unprivileged containers without requiring Cgroup v2 and without the `KubeletInUserNamespace` feature gate. It does this by exposing specially crafted `/proc` and `/sys` filesystems inside the container plus several other advanced OS virtualization techniques.

Running Rootless Kubernetes directly on a host

Note: This section links to third party projects that provide functionality required by Kubernetes. The Kubernetes project authors aren't responsible for these projects, which are listed alphabetically. To add a project to this list, read the [content guide](#) before submitting a change. [More information.](#)

K3s

[K3s](#) experimentally supports rootless mode.

See [Running K3s with Rootless mode](#) for the usage.

Usernetes

[Usernetes](#) is a reference distribution of Kubernetes that can be installed under `$HOME` directory without the root privilege.

Usernetes supports both containerd and CRI-O as CRI runtimes. Usernetes supports multi-node clusters using Flannel (VXLAN).

See [the Usernetes repo](#) for the usage.

Manually deploy a node that runs the kubelet in a user namespace

This section provides hints for running Kubernetes in a user namespace manually.

Note: This section is intended to be read by developers of Kubernetes

distributions, not by end users.

Creating a user namespace

The first step is to create a [user namespace](#).

If you are trying to run Kubernetes in a user-namespaced container such as Rootless Docker/Podman or LXC/LXD, you are all set, and you can go to the next subsection.

Otherwise you have to create a user namespace by yourself, by calling `unshare(2)` with `CLONE_NEWUSER`.

A user namespace can be also unshared by using command line tools such as:

- [unshare\(1\)](#)
- [RootlessKit](#)
- [become-root](#)

After unsharing the user namespace, you will also have to unshare other namespaces such as mount namespace.

You do *not* need to call `chroot()` nor `pivot_root()` after unsharing the mount namespace, however, you have to mount writable filesystems on several directories *in* the namespace.

At least, the following directories need to be writable *in* the namespace (not *outside* the namespace):

- `/etc`
- `/run`
- `/var/logs`
- `/var/lib/kubelet`
- `/var/lib/cni`
- `/var/lib/containerd` (for containerd)
- `/var/lib/containers` (for CRI-O)

Creating a delegated cgroup tree

In addition to the user namespace, you also need to have a writable cgroup tree with cgroup v2.

Note: Kubernetes support for running Node components in user namespaces requires cgroup v2. Cgroup v1 is not supported.

If you are trying to run Kubernetes in Rootless Docker/Podman or LXC/LXD on a systemd-based host, you are all set.

Otherwise you have to create a systemd unit with `Delegate=yes` property to delegate a cgroup tree with writable permission.

On your node, systemd must already be configured to allow delegation; for more details, see [cgroup v2](#) in the Rootless Containers documentation.

Configuring network

Note: This section links to third party projects that provide functionality required by Kubernetes. The Kubernetes project authors aren't responsible for these projects, which are listed alphabetically. To

add a project to this list, read the [content guide](#) before submitting a change. [More information.](#)

The network namespace of the Node components has to have a non-loopback interface, which can be for example configured with [slirp4netns](#), [VPNKit](#), or [lxc-user-nic\(1\)](#).

The network namespaces of the Pods can be configured with regular CNI plugins. For multi-node networking, Flannel (VXLAN, 8472/UDP) is known to work.

Ports such as the kubelet port (10250/TCP) and `NodePort` service ports have to be exposed from the Node network namespace to the host with an external port forwarder, such as RootlessKit, slirp4netns, or [socat\(1\)](#).

You can use the port forwarder from K3s. See [Running K3s in Rootless Mode](#) for more details. The implementation can be found in [the `pkg/rootlessports` package](#) of k3s.

Configuring CRI

The kubelet relies on a container runtime. You should deploy a container runtime such as containerd or CRI-O and ensure that it is running within the user namespace before the kubelet starts.

[containerd](#) [CRI-O](#)

Running CRI plugin of containerd in a user namespace is supported since containerd 1.4.

Running containerd within a user namespace requires the following configurations.

```
version = 2

[plugins."io.containerd.grpc.v1.cri"]
# Disable AppArmor
 disable_apparmor = true
# Ignore an error during setting oom_score_adj
 restrict_oom_score_adj = true
# Disable hugetlb cgroup v2 controller (because systemd does not support delegating
 disable_hugetlb_controller = true

[plugins."io.containerd.grpc.v1.cri".containerd]
# Using non-fuse overlayfs is also possible for kernel >= 5.11, but requires SELinux
 snapshotter = "fuse-overlayfs"

[plugins."io.containerd.grpc.v1.cri".containerd.runtimes.runc.options]
# We use cgroupfs that is delegated by systemd, so we do not use SystemdCgroup drive
# (unless you run another systemd in the namespace)
 SystemdCgroup = false
```

The default path of the configuration file is `/etc/containerd/config.toml`. The path can be specified with `containerd -c /path/to/containerd/config.toml`.

Configuring kubelet

Running kubelet in a user namespace requires the following configuration:

```

apiVersion: kubelet.config.k8s.io/v1beta1
kind: KubeletConfiguration
featureGates:
  KubeletInUserNamespace: true
  # We use cgroupfs that is delegated by systemd, so we do not use "systemd"
  # (unless you run another systemd in the namespace)
  cgroupDriver: "cgroupfs"

```

When the `KubeletInUserNamespace` feature gate is enabled, the kubelet ignores errors that may happen during setting the following sysctl values on the node.

- `vm.overcommit_memory`
- `vm.panic_on_oom`
- `kernel.panic`
- `kernel.panic_on_oops`
- `kernel.keys.root_maxkeys`
- `kernel.keys.root_maxbytes` .

Within a user namespace, the kubelet also ignores any error raised from trying to open `/dev/kmsg`. This feature gate also allows kube-proxy to ignore an error during setting `RLIMIT_NOFILE`.

The `KubeletInUserNamespace` feature gate was introduced in Kubernetes v1.22 with "alpha" status.

Running kubelet in a user namespace without using this feature gate is also possible by mounting a specially crafted proc filesystem (as done by [Sysbox](#)), but not officially supported.

Configuring kube-proxy

Running kube-proxy in a user namespace requires the following configuration:

```

apiVersion: kubeproxy.config.k8s.io/v1alpha1
kind: KubeProxyConfiguration
mode: "iptables" # or "userspace"
conntrack:
  # Skip setting sysctl value "net.netfilter.nf_conntrack_max"
  maxPerCore: 0
  # Skip setting "net.netfilter.nf_conntrack_tcp_timeout_established"
  tcpEstablishedTimeout: 0s
  # Skip setting "net.netfilter.nf_conntrack_tcp_timeout_close"
  tcpCloseWaitTimeout: 0s

```

Caveats

- Most of "non-local" volume drivers such as `nfs` and `iscsi` do not work. Local volumes like `local`, `hostPath`, `emptyDir`, `configMap`, `secret`, and `downwardAPI` are known to work.
- Some CNI plugins may not work. Flannel (VXLAN) is known to work.

For more on this, see the [Caveats and Future work](#) page on the rootlesscontaine.rs website.

See Also

- [rootlesscontaine.rs](#)
- [Rootless Containers 2020 \(KubeCon NA 2020\)](#)
- [Running kind with Rootless Docker](#)
- [Usernetes](#)
- [Running K3s with rootless mode](#)
- [KEP-2033: Kubelet-in-UserNS \(aka Rootless mode\)](#)

2.32 - Safely Drain a Node

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

Before you begin

This task assumes that you have met the following prerequisites:

1. You do not require your applications to be highly available during the node drain, or
2. You have read about the [PodDisruptionBudget](#) concept, and have [configured PodDisruptionBudgets](#) for applications that need them.

(Optional) Configure a disruption budget

To ensure that your workloads remain available during maintenance, you can configure a [PodDisruptionBudget](#).

If availability is important for any applications that run or could run on the node(s) that you are draining, [configure a PodDisruptionBudgets](#) first and then continue following this guide.

It is recommended to set `AlwaysAllow` [Unhealthy Pod Eviction Policy](#) to your PodDisruptionBudgets to support eviction of misbehaving applications during a node drain. The default behavior is to wait for the application pods to become [healthy](#) before the drain can proceed.

Use `kubectl drain` to remove a node from service

You can use `kubectl drain` to safely evict all of your pods from a node before you perform maintenance on the node (e.g. kernel upgrade, hardware maintenance, etc.). Safe evictions allow the pod's containers to [gracefully terminate](#) and will respect the PodDisruptionBudgets you have specified.

Note: By default `kubectl drain` ignores certain system pods on the node that cannot be killed; see the [kubectl drain](#) documentation for more details.

When `kubectl drain` returns successfully, that indicates that all of the pods (except the ones excluded as described in the previous paragraph) have been safely evicted (respecting the desired graceful termination period, and respecting the PodDisruptionBudget you have defined). It is then safe to bring down the node by powering down its physical machine or, if running on a cloud platform, deleting its virtual machine.

Note:

If any new Pods tolerate the `node.kubernetes.io/unschedulable` taint, then those Pods might be scheduled to the node you have drained. Avoid tolerating that taint other than for DaemonSets.

If you or another API user directly set the [nodeName](#) field for a Pod (bypassing the scheduler), then the Pod is bound to the specified node and will run there, even though you have drained that node and marked it unschedulable.

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 --ignore-daemonsets <node name>
```

If there are pods managed by a DaemonSet, you will need to specify `--ignore-daemonsets` with `kubectl` to successfully drain the node. The `kubectl drain` subcommand on its own does not actually drain a node of its DaemonSet pods: the DaemonSet controller (part of the control plane) immediately replaces missing Pods with new equivalent Pods. The DaemonSet controller also creates Pods that ignore unschedulable taints, which allows the new Pods to launch onto a node that you are draining.

Once it returns (without giving an error), you can power down the node (or equivalently, if on a cloud platform, delete the virtual machine backing the node). If you leave the node in the cluster during the maintenance operation, you need to run

```
kubectl uncordon <node name>
```

afterwards to tell Kubernetes that it can resume scheduling new pods onto the node.

Draining multiple nodes in parallel

The `kubectl drain` command should only be issued to a single node at a time. However, you can run multiple `kubectl drain` commands for different nodes in parallel, in different terminals or in the background. Multiple drain commands running concurrently will still respect the `PodDisruptionBudget` you specify.

For example, if you have a StatefulSet with three replicas and have set a `PodDisruptionBudget` for that set specifying `minAvailable: 2`, `kubectl drain` only evicts a pod from the StatefulSet if all three replicas pods are [healthy](#); if then you issue multiple drain commands in parallel, Kubernetes respects the `PodDisruptionBudget` and ensures that only 1 (calculated as `replicas - minAvailable`) Pod is unavailable at any given time. Any drains that would cause the number of [healthy](#) 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.

For more information, see [API-initiated eviction](#).

What's next

- Follow steps to protect your application by [configuring a Pod Disruption Budget](#).

2.33 - Securing a Cluster

This document covers topics related to protecting a cluster from accidental or malicious access and provides recommendations on overall security.

Before you begin

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

To check the version, enter `kubectl version`.

Controlling access to the Kubernetes API

As Kubernetes is entirely API-driven, controlling and limiting who can access the cluster and what actions they are allowed to perform is the first line of defense.

Use Transport Layer Security (TLS) for all API traffic

Kubernetes expects that all API communication in the cluster is encrypted by default with TLS, and the majority of installation methods will allow the necessary certificates to be created and distributed to the cluster components. Note that some components and installation methods may enable local ports over HTTP and administrators should familiarize themselves with the settings of each component to identify potentially unsecured traffic.

API Authentication

Choose an authentication mechanism for the API servers to use that matches the common access patterns when you install a cluster. For instance, small, single-user clusters may wish to use a simple certificate or static Bearer token approach. Larger clusters may wish to integrate an existing OIDC or LDAP server that allow users to be subdivided into groups.

All API clients must be authenticated, even those that are part of the infrastructure like nodes, proxies, the scheduler, and volume plugins. These clients are typically [service accounts](#) or use x509 client certificates, and they are created automatically at cluster startup or are setup as part of the cluster installation.

Consult the [authentication reference document](#) for more information.

API Authorization

Once authenticated, every API call is also expected to pass an authorization check. Kubernetes ships an integrated [Role-Based Access Control \(RBAC\)](#) component that matches an incoming user or group to a set of permissions bundled into roles. These permissions combine verbs (get, create, delete) with resources (pods, services, nodes) and can be namespace-scoped 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 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 run 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.

You can configure [Pod security admission](#) to enforce use of a particular [Pod Security Standard](#) in a namespace, or to detect breaches.

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 apply the **Baseline** or **Restricted** Pod Security Standard.

Preventing containers from loading unwanted kernel modules

The Linux kernel automatically loads kernel modules from disk if needed in certain circumstances, such as when a piece of hardware is attached or a filesystem is mounted. Of particular relevance to Kubernetes, even unprivileged processes can cause certain network-protocol-related kernel modules to be loaded, just by creating a socket of the appropriate type. This may allow an attacker to exploit a security hole in a kernel module that the administrator assumed was not in use.

To prevent specific modules from being automatically loaded, you can uninstall them from the node, or add rules to block them. On most Linux distributions, you can do that by creating a file such as

`/etc/modprobe.d/kubernetes-blacklist.conf` with contents like:

```
# DCCP is unlikely to be needed, has had multiple serious
# vulnerabilities, and is not well-maintained.
blacklist dccp

# SCTP is not used in most Kubernetes clusters, and has also had
# vulnerabilities in the past.
blacklist sctp
```

To block module loading more generically, you can use a Linux Security Module (such as SELinux) to completely deny the `module_request` permission to containers, preventing the kernel from loading modules for containers under any circumstances. (Pods would still be able to use modules that had been loaded manually, or modules that were loaded by the kernel on behalf of some more-privileged process.)

Restricting network access

The [network policies](#) for a namespace allows application authors to restrict which pods in other namespaces may access pods and ports within their namespaces. Many of the supported [Kubernetes networking providers](#) now respect network policy.

Quota and limit ranges can also be used to control whether users may request node ports or load-balanced services, which on many clusters can control whether those users' applications are visible outside of the cluster.

Additional protections may be available that control network rules on a per-plugin or per-environment basis, such as per-node firewalls, physically separating cluster nodes to prevent cross talk, or advanced networking policy.

Restricting cloud metadata API access

Cloud platforms (AWS, Azure, GCE, etc.) often expose metadata services locally to instances. By default these APIs are accessible by pods running on an instance and can contain cloud credentials for that node, or

provisioning data such as kubelet credentials. These credentials can be used to escalate within the cluster or to other cloud services under the same account.

When running Kubernetes on a cloud platform, limit permissions given to instance credentials, use [network policies](#) to restrict pod access to the metadata API, and avoid using provisioning data to deliver secrets.

Controlling which nodes pods may access

By default, there are no restrictions on which nodes may run a pod. Kubernetes offers a [rich set of policies for controlling placement of pods onto nodes](#) and the [taint-based pod placement and eviction](#) that are available to end users. For many clusters use of these policies to separate workloads can be a convention that authors adopt or enforce via tooling.

As an administrator, a beta admission plugin `PodNodeSelector` can be used to force pods within a namespace to default or require a specific node selector, and if end users cannot alter namespaces, this can strongly limit the placement of all of the pods in a specific workload.

Protecting cluster components from compromise

This section describes some common patterns for protecting clusters from compromise.

Restrict access to etcd

Write access to the etcd backend for the API is equivalent to gaining root on the entire cluster, and read access can be used to escalate fairly quickly. Administrators should always use strong credentials from the API servers to their etcd server, such as mutual auth via TLS client certificates, and it is often recommended to isolate the etcd servers behind a firewall that only the API servers may access.

Caution: Allowing other components within the cluster to access the master etcd instance with read or write access to the full keyspace is equivalent to granting cluster-admin access. Using separate etcd instances for non-master components or using etcd ACLs to restrict read and write access to a subset of the keyspace is strongly recommended.

Enable audit logging

The [audit logger](#) is a beta feature that records actions taken by the API for later analysis in the event of a compromise. It is recommended to enable audit logging and archive the audit file on a secure server.

Restrict access to alpha or beta features

Alpha and beta Kubernetes features are in active development and may have limitations or bugs that result in security vulnerabilities. Always assess the value an alpha or beta feature may provide against the possible risk to your security posture. When in doubt, disable features you do not use.

Rotate infrastructure credentials frequently

The shorter the lifetime of a secret or credential the harder it is for an attacker to make use of that credential. Set short lifetimes on certificates and automate their rotation. Use an authentication provider that can control how long issued tokens are available and use short lifetimes where possible. If you use service-account tokens in external integrations, plan to rotate those tokens frequently. For example, once the bootstrap phase is complete, a bootstrap token used for setting up nodes should be revoked or its authorization removed.

Review third party integrations before enabling them

Many third party integrations to Kubernetes may alter the security profile of your cluster. When enabling an integration, always review the permissions that an extension requests before granting it access. For example, many security integrations may request access to view all secrets on your cluster which is effectively making that component a cluster admin. When in doubt, restrict the integration to functioning in a single namespace if possible.

Components that create pods may also be unexpectedly powerful if they can do so inside namespaces like the `kube-system` namespace, because those pods can gain access to service account secrets or run with elevated permissions if those service accounts are granted access to permissive [PodSecurityPolicies](#).

If you use [Pod Security admission](#) and allow any component to create Pods within a namespace that permits privileged Pods, those Pods may be able to escape their containers and use this widened access to elevate their privileges.

You should not allow untrusted components to create Pods in any system namespace (those with names that start with `kube-`) nor in any namespace where that access grant allows the possibility of privilege escalation.

Encrypt secrets at rest

In general, the etcd database will contain any information accessible via the Kubernetes API and may grant an attacker significant visibility into the state of your cluster. Always encrypt your backups using a well reviewed backup and encryption solution, and consider using full disk encryption where possible.

Kubernetes supports optional [encryption at rest](#) for information in the Kubernetes API. This lets you ensure that when Kubernetes stores data for objects (for example, `Secret` or `ConfigMap` objects), the API server writes an encrypted representation of the object. That encryption means that even someone who has access to etcd backup data is unable to view the content of those objects. In Kubernetes 1.28 you can also encrypt custom resources; encryption-at-rest for extension APIs defined in `CustomResourceDefinitions` was added to Kubernetes as part of the v1.26 release.

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.

What's next

- [Security Checklist](#) for additional information on Kubernetes security guidance.

2.34 - Set Kubelet Parameters Via A Configuration File

A subset of the kubelet's configuration parameters may be set via an on-disk config file, as a substitute for command-line flags.

Providing parameters via a config file is the recommended approach because it simplifies node deployment and configuration management.

Create the config file

The subset of the kubelet's configuration that can be configured via a file is defined by the [KubeletConfiguration](#) struct.

The configuration file must be a JSON or YAML representation of the parameters in this struct. Make sure the kubelet has read permissions on the file.

Here is an example of what this file might look like:

```
apiVersion: kubelet.config.k8s.io/v1beta1
kind: KubeletConfiguration
address: "192.168.0.8"
port: 20250
serializeImagePulls: false
evictionHard:
  memory.available: "100Mi"
  nodefs.available: "10%"
  nodefs.inodesFree: "5%"
  imagefs.available: "15%"
```

In this example, the kubelet is configured with the following settings:

1. address : The kubelet will serve on IP address 192.168.0.8 .
2. port : The kubelet will serve on port 20250 .
3. serializeImagePulls : Image pulls will be done in parallel.
4. evictionHard : The kubelet will evict Pods under one of the following conditions:
 - o When the node's available memory drops below 100MiB.
 - o When the node's main filesystem's available space is less than 10%.
 - o When the image filesystem's available space is less than 15%.
 - o When more than 95% of the node's main filesystem's inodes are in use.

Note: In the example, by changing the default value of only one parameter for evictionHard, the default values of other parameters will not be inherited and will be set to zero. In order to provide custom values, you should provide all the threshold values respectively.

The `imagefs` is an optional filesystem that container runtimes use to store container images and container writable layers.

Start a kubelet process configured via the config file

Note: If you use kubeadm to initialize your cluster, use the kubelet-config while creating your cluster with `kubeadm init`. See [configuring kubelet using kubeadm](#) for details.

Start the kubelet with the `--config` flag set to the path of the kubelet's config file. The kubelet will then load its config from this file.

Note that command line flags which target the same value as a config file will override that value. This helps ensure backwards compatibility with the command-line API.

Note that relative file paths in the kubelet config file are resolved relative to the location of the kubelet config file, whereas relative paths in command line flags are resolved relative to the kubelet's current working directory.

Note that some default values differ between command-line flags and the kubelet config file. If `--config` is provided and the values are not specified via the command line, the defaults for the `KubeletConfiguration` version apply. In the above example, this version is `kubelet.config.k8s.io/v1beta1`.

Drop-in directory for kubelet configuration files

As of Kubernetes v1.28.0, the kubelet has been extended to support a drop-in configuration directory. The location of it can be specified with `--config-dir` flag, and it defaults to `""`, or disabled, by default.

You can only set `--config-dir` if you set the environment variable `KUBELET_CONFIG_DROPIN_DIR_ALPHA` for the kubelet process (the value of that variable does not matter). For Kubernetes v1.28, the kubelet returns an error if you specify `--config-dir` without that variable set, and startup fails. You cannot specify the drop-in configuration directory using the kubelet configuration file; only the CLI argument `--config-dir` can set it.

One can use the kubelet configuration directory in a similar way to the kubelet config file.

Note: The suffix of a valid kubelet drop-in configuration file must be `.conf`. For instance: `99-kubelet-address.conf`

For instance, you may want a baseline kubelet configuration for all nodes, but you may want to customize the `address` field. This can be done as follows:

Main kubelet configuration file contents:

```
apiVersion: kubelet.config.k8s.io/v1beta1
kind: KubeletConfiguration
port: 20250
serializeImagePulls: false
evictionHard:
  memory.available: "200Mi"
```

Contents of a file in `--config-dir` directory:

```
apiVersion: kubelet.config.k8s.io/v1beta1
kind: KubeletConfiguration
address: "192.168.0.8"
```

On startup, the kubelet merges configuration from:

- Command line arguments (lowest precedence).
- the kubelet configuration
- Drop-in configuration files, according to sort order.
- Feature gates specified over the command line (highest precedence).

This produces the same outcome as if you used the [single configuration file](#) used in the earlier example.

What's next

- Learn more about kubelet configuration by checking the [KubeletConfiguration](#) reference.

2.35 - Share a Cluster with Namespaces

This page shows how to view, work in, and delete namespaces. The page also shows how to use Kubernetes namespaces to subdivide your cluster.

Before you begin

- Have an [existing Kubernetes cluster](#).
- You have a basic understanding of Kubernetes [Pods](#), [Services](#), and [Deployments](#).

Viewing namespaces

List the current namespaces in a cluster using:

```
kubectl get namespaces
```

NAME	STATUS	AGE
default	Active	11d
kube-node-lease	Active	11d
kube-public	Active	11d
kube-system	Active	11d

Kubernetes starts with four initial namespaces:

- **default** The default namespace for objects with no other namespace
- **kube-node-lease** This namespace holds [Lease](#) objects associated with each node. Node leases allow the kubelet to send [heartbeats](#) so that the control plane can detect node failure.
- **kube-public** This namespace is created automatically and is readable by all users (including those not authenticated). This namespace is mostly reserved for cluster usage, in case that some resources should be visible and readable publicly throughout the whole cluster. The public aspect of this namespace is only a convention, not a requirement.
- **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

For more details, see [Namespace](#) in the API reference.

Creating a new namespace

Note: Avoid creating namespace with prefix `kube-`, since it is reserved for Kubernetes system namespaces.

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
```

Alternatively, you can create namespace using below command:

```
kubectl create namespace <insert-namespace-name-here>
```

The name of your namespace must be a valid [DNS label](#).

There's an optional field `finalizers`, which allows observables to purge resources whenever the namespace is deleted. Keep in mind that if you specify a nonexistent finalizer, the namespace will be created but will get stuck in the `Terminating` state if the user tries to delete it.

More information on `finalizers` can be found in the namespace [design doc](#).

Deleting a namespace

Delete a namespace with

```
kubectl delete namespaces <insert-some-namespace-name>
```

Warning: This deletes *everything* under the namespace!

This delete is asynchronous, so for a time you will see the namespace in the `Terminating` state.

Subdividing your cluster using Kubernetes namespaces

By default, a Kubernetes cluster will instantiate a default namespace when provisioning the cluster to hold the default set of Pods, Services, and Deployments used by the cluster.

Assuming you have a fresh cluster, you can introspect the available namespaces by doing the following:

```
kubectl get namespaces
```

NAME	STATUS	AGE
default	Active	13m

Create new namespaces

For this exercise, we will create two additional Kubernetes namespaces to hold our content.

In a scenario where an organization is using a shared Kubernetes cluster for development and production use cases:

- The development team would like to maintain a space in the cluster where they can get a view on the list of Pods, Services, and Deployments they use to build and run their application. In this space, Kubernetes resources come and go, and the restrictions on who can or cannot modify resources are relaxed to enable agile development.
- The operations team would like to maintain a space in the cluster where they can enforce strict procedures on who can or cannot manipulate the set of Pods, Services, and Deployments that run the production site.

One pattern this organization could follow is to partition the Kubernetes cluster into two namespaces: `development` and `production`. Let's create two new namespaces to hold our work.

Create the `development` namespace using kubectl:

```
kubectl create -f https://k8s.io/examples/admin/namespace-dev.json
```

And then let's create the `production` namespace using kubectl:

```
kubectl create -f https://k8s.io/examples/admin/namespace-prod.json
```

To be sure things are right, list all of the namespaces in our cluster.

```
kubectl get namespaces --show-labels
```

NAME	STATUS	AGE	LABELS
default	Active	32m	<none>
development	Active	29s	name=development
production	Active	23s	name=production

Create pods in each namespace

A Kubernetes namespace provides the scope for Pods, Services, and Deployments in the cluster. Users interacting with one namespace do not see the content in another namespace. To demonstrate this, let's spin up a simple Deployment and Pods in the `development` namespace.

```
kubectl create deployment snowflake \
--image=registry.k8s.io/serve_hostname \
-n=development --replicas=2
```

We have created a deployment whose replica size is 2 that is running the pod called `snowflake` with a basic container that serves the hostname.

```
kubectl get deployment -n=development
```

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

```
kubectl get pods -l app=snowflake -n=development
```

NAME	READY	STATUS	RESTARTS	AGE
snowflake-3968820950-9dgr8	1/1	Running	0	2m
snowflake-3968820950-vgc4n	1/1	Running	0	2m

And this is great, developers are able to do what they want, and they do not have to worry about affecting content in the `production` namespace.

Let's switch to the `production` namespace and show how resources in one namespace are hidden from the other. The `production` namespace should be empty, and the following commands should return nothing.

```
kubectl get deployment -n=production  
kubectl get pods -n=production
```

Production likes to run cattle, so let's create some cattle pods.

```
kubectl create deployment cattle --image=registry.k8s.io/serve_hostname  
kubectl scale deployment cattle --replicas=5 -n=production  
  
kubectl get deployment -n=production
```

NAME	READY	UP-TO-DATE	AVAILABLE	AGE
cattle	5/5	5	5	10s

```
kubectl get pods -l app=cattle -n=production
```

NAME	READY	STATUS	RESTARTS	AGE
cattle-2263376956-41xy6	1/1	Running	0	34s
cattle-2263376956-kw466	1/1	Running	0	34s
cattle-2263376956-n4v97	1/1	Running	0	34s
cattle-2263376956-p5p3i	1/1	Running	0	34s
cattle-2263376956-sxpth	1/1	Running	0	34s

At this point, it should be clear that the resources users create in one namespace are hidden from the other namespace.

As the policy support in Kubernetes evolves, we will extend this scenario to show how you can provide different authorization rules for each namespace.

Understanding the motivation for using namespaces

A single cluster should be able to satisfy the needs of multiple users or groups of users (henceforth in this document a *user community*).

Kubernetes *namespaces* help different projects, teams, or customers to share a Kubernetes cluster.

It does this by providing the following:

1. A scope for [names](#).
2. A mechanism to attach authorization and policy to a subsection of the cluster.

Use of multiple namespaces is optional.

Each user community wants to be able to work in isolation from other communities. Each user community has its own:

1. resources (pods, services, replication controllers, etc.)
2. policies (who can or cannot perform actions in their community)
3. constraints (this community is allowed this much quota, etc.)

A cluster operator may create a Namespace for each unique user community.

The Namespace provides a unique scope for:

1. named resources (to avoid basic naming collisions)
2. delegated management authority to trusted users
3. ability to limit community resource consumption

Use cases include:

1. As a cluster operator, I want to support multiple user communities on a single cluster.
2. As a cluster operator, I want to delegate authority to partitions of the cluster to trusted users in those communities.
3. As a cluster operator, I want to limit the amount of resources each community can consume in order to limit the impact to other communities using the cluster.
4. As a cluster user, I want to interact with resources that are pertinent to my user community in isolation of what other user communities are doing on the cluster.

Understanding namespaces and DNS

When you create a [Service](#), it creates a corresponding [DNS entry](#). This entry is of the form `<service-name>.<namespace-name>.svc.cluster.local`, which means that if a container uses `<service-name>` it will resolve to the service which is local to a namespace. This is useful for using the same configuration across multiple namespaces such as Development, Staging and Production. If you want to reach across namespaces, you need to use the fully qualified domain name (FQDN).

What's next

- Learn more about [setting the namespace preference](#).
- Learn more about [setting the namespace for a request](#)
- See [namespaces design](#).

2.36 - Upgrade A Cluster

This page provides an overview of the steps you should follow to upgrade a Kubernetes cluster.

The way that you upgrade a cluster depends on how you initially deployed it and on any subsequent changes.

At a high level, the steps you perform are:

- Upgrade the [control plane](#)
- Upgrade the nodes in your cluster
- Upgrade clients such as [kubectl](#)
- Adjust manifests and other resources based on the API changes that accompany the new Kubernetes version

Before you begin

You must have an existing cluster. This page is about upgrading from Kubernetes 1.27 to Kubernetes 1.28. If your cluster is not currently running Kubernetes 1.27 then please check the documentation for the version of Kubernetes that you plan to upgrade to.

Upgrade approaches

kubeadm

If your cluster was deployed using the `kubeadm` tool, refer to [Upgrading kubeadm clusters](#) for detailed information on how to upgrade the cluster.

Once you have upgraded the cluster, remember to [install the latest version of kubectl](#).

Manual deployments

Caution: These steps do not account for third-party extensions such as network and storage plugins.

You should manually update the control plane following this sequence:

- etcd (all instances)
- kube-apiserver (all control plane hosts)
- kube-controller-manager
- kube-scheduler
- cloud controller manager, if you use one

At this point you should [install the latest version of kubectl](#).

For each node in your cluster, [drain](#) that node and then either replace it with a new node that uses the 1.28 kubelet, or upgrade the kubelet on that node and bring the node back into service.

Caution: Draining nodes before upgrading kubelet ensures that pods are re-admitted and containers are re-created, which may be necessary to resolve some security issues or other important bugs.

Other deployments

Refer to the documentation for your cluster deployment tool to learn the recommended set up steps for maintenance.

Post-upgrade tasks

Switch your cluster's storage API version

The objects that are serialized into etcd for a cluster's internal representation of the Kubernetes resources active in the cluster are written using a particular version of the API.

When the supported API changes, these objects may need to be rewritten in the newer API. Failure to do this will eventually result in resources that are no longer decodable or usable by the Kubernetes API server.

For each affected object, fetch it using the latest supported API and then write it back also using the latest supported API.

Update manifests

Upgrading to a new Kubernetes version can provide new APIs.

You can use `kubectl convert` command to convert manifests between different API versions. For example:

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

The `kubectl` tool replaces the contents of `pod.yaml` with a manifest that sets `kind` to Pod (unchanged), but with a revised `apiVersion`.

Device Plugins

If your cluster is running device plugins and the node needs to be upgraded to a Kubernetes release with a newer device plugin API version, device plugins must be upgraded to support both version before the node is upgraded in order to guarantee that device allocations continue to complete successfully during the upgrade.

Refer to [API compatibility](#) and [Kubelet Device Manager API Versions](#) for more details.

2.37 - Use Cascading Deletion in a Cluster

This page shows you how to specify the type of [cascading deletion](#) to use in your cluster during garbage collection.

Before you begin

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

- [Killercoda](#)
- [Play with Kubernetes](#)

You also need to [create a sample Deployment](#) to experiment with the different types of cascading deletion. You will need to recreate the Deployment for each type.

Check owner references on your pods

Check that the `ownerReferences` field is present on your pods:

```
kubectl get pods -l app=nginx --output=yaml
```

The output has an `ownerReferences` field similar to this:

```
apiVersion: v1
...
ownerReferences:
- apiVersion: apps/v1
  blockOwnerDeletion: true
  controller: true
  kind: ReplicaSet
  name: nginx-deployment-6b474476c4
  uid: 4fdcd81c-bd5d-41f7-97af-3a3b759af9a7
...
```

Use foreground cascading deletion

By default, Kubernetes uses [background cascading deletion](#) to delete dependents of an object. You can switch to foreground cascading deletion using either `kubectl` or the Kubernetes API, depending on the Kubernetes version your cluster runs. To check the version, enter `kubectl version`.

You can delete objects using foreground cascading deletion using `kubectl` or the Kubernetes API.

Using `kubectl`

Run the following command:

```
kubectl delete deployment nginx-deployment --cascade=foreground
```

Using the Kubernetes API

1. Start a local proxy session:

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

2. Use `curl` to trigger deletion:

```
curl -X DELETE localhost:8080/apis/apps/v1/namespaces/default/dep  
-d '{"kind":"DeleteOptions","apiVersion":"v1","propagationPol  
-H "Content-Type: application/json"
```

The output contains a `foregroundDeletion` finalizer like this:

```
"kind": "Deployment",  
"apiVersion": "apps/v1",  
"metadata": {  
    "name": "nginx-deployment",  
    "namespace": "default",  
    "uid": "d1ce1b02-cae8-4288-8a53-30e84d8fa505",  
    "resourceVersion": "1363097",  
    "creationTimestamp": "2021-07-08T20:24:37Z",  
    "deletionTimestamp": "2021-07-08T20:27:39Z",  
    "finalizers": [  
        "foregroundDeletion"  
    ]  
    ...
```

Use background cascading deletion

1. [Create a sample Deployment](#).
2. Use either `kubectl` or the Kubernetes API to delete the Deployment, depending on the Kubernetes version your cluster runs. To check the version, enter `kubectl version`.

You can delete objects using background cascading deletion using `kubectl` or the Kubernetes API.

Kubernetes uses background cascading deletion by default, and does so even if you run the following commands without the `--cascade` flag or the `propagationPolicy` argument.

Using `kubectl`

Run the following command:

```
kubectl delete deployment nginx-deployment --cascade=background
```

Using the Kubernetes API

1. Start a local proxy session:

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

2. Use `curl` to trigger deletion:

```
curl -X DELETE localhost:8080/apis/apps/v1/namespaces/default/dep  
-d '{"kind":"DeleteOptions","apiVersion":"v1","propagationPol  
-H "Content-Type: application/json"
```

The output is similar to this:

```
"kind": "Status",  
"apiVersion": "v1",  
...  
"status": "Success",  
"details": {  
    "name": "nginx-deployment",  
    "group": "apps",  
    "kind": "deployments",  
    "uid": "cc9eefb9-2d49-4445-b1c1-d261c9396456"  
}
```

Delete owner objects and orphan dependents

By default, when you tell Kubernetes to delete an object, the [controller](#) also deletes dependent objects. You can make Kubernetes *orphan* these dependents using `kubectl` or the Kubernetes API, depending on the Kubernetes version your cluster runs. To check the version, enter `kubectl version`.

Using `kubectl`

Run the following command:

```
kubectl delete deployment nginx-deployment --cascade=orphan
```

Using the Kubernetes API

1. Start a local proxy session:

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

2. Use `curl` to trigger deletion:

```
curl -X DELETE localhost:8080/apis/apps/v1/namespaces/default/dep  
-d '{"kind":"DeleteOptions","apiVersion":"v1","propagationPol  
-H "Content-Type: application/json"
```

The output contains `orphan` in the `finalizers` field, similar to this:

```
"kind": "Deployment",
"apiVersion": "apps/v1",
"namespace": "default",
"uid": "6f577034-42a0-479d-be21-78018c466f1f",
"creationTimestamp": "2021-07-09T16:46:37Z",
"deletionTimestamp": "2021-07-09T16:47:08Z",
"deletionGracePeriodSeconds": 0,
"finalizers": [
    "orphan"
],
...
```

You can check that the Pods managed by the Deployment are still running:

```
kubectl get pods -l app=nginx
```

What's next

- Learn about [owners and dependents](#) in Kubernetes.
- Learn about Kubernetes [finalizers](#).
- Learn about [garbage collection](#).

2.38 - 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. In Kubernetes 1.28 there are two versions of KMS at-rest encryption. You should use KMS v2 if feasible because KMS v1 is deprecated (since Kubernetes v1.28). However, you should also read and observe the **Caution** notices in this page that highlight specific cases when you must not use KMS v2. KMS v2 offers significantly better performance characteristics than KMS v1.

Before you begin

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

- [Killercoda](#)
- [Play with Kubernetes](#)

The version of Kubernetes that you need depends on which KMS API version you have selected. Kubernetes recommends using KMS v2.

- If you selected KMS API v2, you should use Kubernetes v1.28 (if you are running a different version of Kubernetes that also supports the v2 KMS API, switch to the documentation for that version of Kubernetes).
- If you selected KMS API v1 to support clusters prior to version v1.27 or if you have a legacy KMS plugin that only supports KMS v1, any supported Kubernetes version will work. This API is deprecated as of Kubernetes v1.28. Kubernetes does not recommend the use of this API.

To check the version, enter `kubectl version`.

KMS v1

FEATURE STATE: [Kubernetes v1.28 \[deprecated\]](#)

- Kubernetes version 1.10.0 or later is required
- Your cluster must use etcd v3 or later

KMS v2

FEATURE STATE: [Kubernetes v1.27 \[beta\]](#)

- For version 1.25 and 1.26, enabling the feature via `kube-apiserver` feature gate is required. Set `--feature-gates=KMSv2=true` to configure a KMS v2 provider. For environments where all API servers are running version 1.28 or later, and you do not require the ability to downgrade to Kubernetes v1.27, you can enable the `KMSv2KDF` feature gate (a beta feature) for more robust data encryption key generation. The Kubernetes project recommends enabling KMS v2 KDF if those preconditions are met.
- Your cluster must use etcd v3 or later

Caution:

The KMS v2 API and implementation changed in incompatible ways in-between the alpha release in v1.25 and the beta release in v1.27. Attempting to upgrade from old versions with the alpha feature enabled will result in data loss.

Running mixed API server versions with some servers at v1.27, and others at v1.28 *with the `KMSv2KDF` feature gate enabled* is **not supported** - and is likely to result in data loss.

The KMS encryption provider uses an envelope encryption scheme to encrypt data in etcd. The data is encrypted using a data encryption key (DEK). The DEKs are encrypted with a key encryption key (KEK) that is stored and managed in a remote KMS.

With KMS v1, a new DEK is generated for each encryption.

With KMS v2, there are two ways for the API server to generate a DEK. Kubernetes defaults to generating a new DEK at API server startup, which is then reused for resource encryption. However, if you use KMS v2 *and enable the `KMSv2KDF` feature gate*, then Kubernetes instead generates a new DEK **per encryption**: the API server uses a *key derivation function* to generate single use data encryption keys from a secret seed combined with some random data. Whichever approach you configure, the DEK or seed is also rotated whenever the KEK is rotated (see [Understanding key_id](#) and [Key Rotation](#) section below for more details).

The KMS provider uses gRPC to communicate with a specific KMS plugin over a UNIX domain socket. The KMS plugin, which is implemented as a gRPC server and deployed on the same host(s) as the Kubernetes control plane, is responsible for all communication with the remote KMS.

Caution:

If you are running virtual machine (VM) based nodes that leverage VM state store with this feature, using KMS v2 is **insecure** and an information security risk unless you also explicitly enable the `KMSv2KDF feature gate`.

With KMS v2, the API server uses AES-GCM with a 12 byte nonce (8 byte atomic counter and 4 bytes random data) for encryption. The following issues could occur if the VM is saved and restored:

1. The counter value may be lost or corrupted if the VM is saved in an inconsistent state or restored improperly. This can lead to a situation where the same counter value is used twice, resulting in the same nonce being used for two different messages.
2. If the VM is restored to a previous state, the counter value may be set back to its previous value, resulting in the same nonce being used again.

Although both of these cases are partially mitigated by the 4 byte random nonce, this can compromise the security of the encryption.

If you have enabled the `KMSv2KDF feature gate` and are using KMS v2 (not KMS v1), the API server generates single use data encryption keys from a secret seed. This eliminates the need for a counter based nonce while avoiding nonce collision concerns. It also removes any specific concerns with using KMS v2 and VM state store.

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:

KMS v1

- `apiVersion` : API Version for KMS provider. Leave this value empty or set it to `v1`.
- `name` : Display name of the KMS plugin. Cannot be changed once set.
- `endpoint` : Listen address of the gRPC server (KMS plugin). The endpoint is a UNIX domain socket.
- `cachesize` : Number of data encryption keys (DEKs) to be cached in the clear. When cached, DEKs can be used without another call to the KMS; whereas DEKs that are not cached require a call to the KMS to unwrap.
- `timeout` : How long should `kube-apiserver` wait for `kms-plugin` to respond before returning an error (default is 3 seconds).

KMS v2

- `apiVersion` : API Version for KMS provider. Set this to `v2`.
- `name` : Display name of the KMS plugin. Cannot be changed once set.
- `endpoint` : Listen address of the gRPC server (KMS plugin). The endpoint is a UNIX domain socket.
- `timeout` : How long should `kube-apiserver` wait for `kms-plugin` to respond before returning an error (default is 3 seconds).

KMS v2 does not support the `cachesize` property. All data encryption keys (DEKs) will be cached in the clear once the server has unwrapped them via a call to the KMS. Once cached, DEKs can be used to perform decryption indefinitely without making a call to the KMS.

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 control plane.

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.

KMS v1

- Using Go: Use the functions and data structures in the stub file: [api.pb.go](#) to develop the gRPC server code

- Using languages other than Go: Use the protoc compiler with the proto file: [api.proto](#) to generate a stub file for the specific language

KMS v2

- Using Go: A high level [library](#) is provided to make the process easier. Low level implementations can use the functions and data structures in the stub file: [api.pb.go](#) to develop the gRPC server code
- Using languages other than Go: Use the protoc compiler with the proto file: [api.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 v1

- 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 `v1beta1`.
- protocol: UNIX domain socket (`unix`)
The plugin is implemented as a gRPC server that listens at UNIX domain socket. The plugin deployment should create a file on the file system to run the gRPC unix domain socket connection. The API server (gRPC client) is configured with the KMS provider (gRPC server) unix domain socket endpoint in order to communicate with it. An abstract Linux socket may be used by starting the endpoint with `/@`, i.e. `unix:///@foo`. Care must be taken when using this type of socket as they do not have concept of ACL (unlike traditional file based sockets). However, they are subject to Linux networking namespace, so will only be accessible to containers within the same pod unless host networking is used.

KMS v2

- KMS plugin version: `v2beta1`
In response to procedure call `status`, a compatible KMS plugin should return `v2beta1` as `StatusResponse.version`, "ok" as `StatusResponse.healthz` and a `key_id` (remote KMS KEK ID) as `StatusResponse.key_id`.
The API server polls the `status` procedure call approximately every minute when everything is healthy, and every 10 seconds when the plugin is not healthy. Plugins must take care to optimize this call as it will be under constant load.
- Encryption
The `EncryptRequest` procedure call provides the plaintext and a UID for logging purposes. The response must include the ciphertext, the `key_id` for the KEK used, and, optionally, any metadata that the KMS plugin needs to aid in future `DecryptRequest` calls (via the

annotations field). The plugin must guarantee that any distinct plaintext results in a distinct response (ciphertext, key_id, annotations) .

If the plugin returns a non-empty annotations map, all map keys must be fully qualified domain names such as example.com . An example use case of annotation is {"kms.example.io/remote-kms-auditid":<audit ID used by the remote KMS>"}

The API server does not perform the EncryptRequest procedure call at a high rate. Plugin implementations should still aim to keep each request's latency at under 100 milliseconds.

- Decryption

The DecryptRequest procedure call provides the (ciphertext, key_id, annotations) from EncryptRequest and a UID for logging purposes. As expected, it is the inverse of the EncryptRequest call. Plugins must verify that the key_id is one that they understand - they must not attempt to decrypt data unless they are sure that it was encrypted by them at an earlier time.

The API server may perform thousands of DecryptRequest procedure calls on startup to fill its watch cache. Thus plugin implementations must perform these calls as quickly as possible, and should aim to keep each request's latency at under 10 milliseconds.

- Understanding key_id and Key Rotation

The key_id is the public, non-secret name of the remote KMS KEK that is currently in use. It may be logged during regular operation of the API server, and thus must not contain any private data. Plugin implementations are encouraged to use a hash to avoid leaking any data. The KMS v2 metrics take care to hash this value before exposing it via the /metrics endpoint.

The API server considers the key_id returned from the status procedure call to be authoritative. Thus, a change to this value signals to the API server that the remote KEK has changed, and data encrypted with the old KEK should be marked stale when a no-op write is performed (as described below). If an EncryptRequest procedure call returns a key_id that is different from status , the response is thrown away and the plugin is considered unhealthy. Thus implementations must guarantee that the key_id returned from status will be the same as the one returned by EncryptRequest . Furthermore, plugins must ensure that the key_id is stable and does not flip-flop between values (i.e. during a remote KEK rotation).

Plugins must not re-use key_id s, even in situations where a previously used remote KEK has been reinstated. For example, if a plugin was using key_id=A , switched to key_id=B , and then went back to key_id=A - instead of reporting key_id=A the plugin should report some derivative value such as key_id=A_001 or use a new value such as key_id=C .

Since the API server polls status about every minute, key_id rotation is not immediate. Furthermore, the API server will coast on the last valid state for about three minutes. Thus if a user wants to take a passive approach to storage migration (i.e. by waiting), they must schedule a migration to occur at $3 + N + M$ minutes after the remote KEK has been rotated (N is how long it takes the plugin to observe the key_id change and M is the desired buffer to allow config changes to be processed - a minimum M of five minutes is recommend). Note that no API server restart is required to perform KEK rotation.

Caution: Because you don't control the number of writes performed with the DEK, the Kubernetes project recommends rotating the KEK at least every 90 days.

- protocol: UNIX domain socket (unix)

The plugin is implemented as a gRPC server that listens at UNIX domain socket. The plugin deployment should create a file on the file system to run the gRPC unix domain socket connection. The API server (gRPC client) is configured with the KMS provider (gRPC server) unix domain socket endpoint in order to communicate with it. An abstract Linux socket may be used by starting the endpoint with `/@` , i.e. `unix:///@foo` . Care must be taken when using this type of socket as they do not have concept of ACL (unlike traditional file based sockets). However, they are subject to Linux networking namespace, so will only be accessible to containers within the same pod unless host networking is used.

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 (KMS v2 makes this process easier by providing a dedicated `annotations` field).

Deploying the KMS plugin

Ensure that the KMS plugin runs on the same host(s) as the Kubernetes API server(s).

Encrypting your data with the KMS provider

To encrypt the data:

1. Create a new `EncryptionConfiguration` file using the appropriate properties for the `kms` provider to encrypt resources like Secrets and ConfigMaps. If you want to encrypt an extension API that is defined in a CustomResourceDefinition, your cluster must be running Kubernetes v1.26 or newer.
2. Set the `--encryption-provider-config` flag on the kube-apiserver to point to the location of the configuration file.
3. `--encryption-provider-config-automatic-reload` boolean argument determines if the file set by `--encryption-provider-config` should be automatically reloaded if the disk contents change. This enables key rotation without API server restarts.
4. Restart your API server.

KMS v1

```

apiVersion: apiserver.config.k8s.io/v1
kind: EncryptionConfiguration
resources:
  - resources:
    - secrets
    - configmaps
    - pandas.awesome.bears.example
providers:
  - kms:
    name: myKmsPluginFoo
    endpoint: unix:///tmp/socketfile.sock
    cachesize: 100
    timeout: 3s
  - kms:
    name: myKmsPluginBar
    endpoint: unix:///tmp/socketfile.sock
    cachesize: 100
    timeout: 3s

```

KMS v2

```

apiVersion: apiserver.config.k8s.io/v1
kind: EncryptionConfiguration
resources:
  - resources:
    - secrets
    - configmaps
    - pandas.awesome.bears.example
providers:
  - kms:
    apiVersion: v2
    name: myKmsPluginFoo
    endpoint: unix:///tmp/socketfile.sock
    timeout: 3s
  - kms:
    apiVersion: v2
    name: myKmsPluginBar
    endpoint: unix:///tmp/socketfile.sock
    timeout: 3s

```

Setting `--encryption-provider-config-automatic-reload` to `true` collapses all health checks to a single health check endpoint. Individual health checks are only available when KMS v1 providers are in use and the encryption config is not auto-reloaded.

The following table summarizes the health check endpoints for each KMS version:

KMS configurations	Without Automatic Reload	With Automatic Reload
KMS v1 only	Individual Healthchecks	Single Healthcheck
KMS v2 only	Single Healthcheck	Single Healthcheck
Both KMS v1 and v2	Individual Healthchecks	Single Healthcheck
No KMS	None	Single Healthcheck

`Single Healthcheck` means that the only health check endpoint is `/healthz/kms-providers`.

Individual Healthchecks means that each KMS plugin has an associated health check endpoint based on its location in the encryption config:
/healthz/kms-provider-0 , /healthz/kms-provider-1 etc.

These healthcheck endpoint paths are hard coded and generated/controlled by the server. The indices for individual healthchecks corresponds to the order in which the KMS encryption config is processed.

At a high level, restarting an API server when a KMS plugin is unhealthy is unlikely to make the situation better. It can make the situation significantly worse by throwing away the API server's DEK cache. Thus the general recommendation is to ignore the API server KMS healthz checks for liveness purposes, i.e. /livez?exclude=kms-providers .

Until the steps defined in [Ensuring all secrets are encrypted](#) are performed, the providers list should end with the identity: {} provider to allow unencrypted data to be read. Once all resources are encrypted, the identity provider should be removed to prevent the API server from honoring unencrypted data.

For details about the EncryptionConfiguration format, please check the [API server encryption API reference](#).

Verifying that the data is encrypted

When encryption at rest is correctly configured, resources are encrypted on write. After restarting your kube-apiserver , any newly created or updated Secret or other resource types configured in

EncryptionConfiguration should be encrypted when stored. To verify, you can use the etcdctl command line program to retrieve the contents of your secret data.

1. Create a new secret called secret1 in the default namespace:

```
kubectl create secret generic secret1 -n default --from-literal=m
```

2. Using the etcdctl command line, read that secret out of etcd:

```
ETCDCTL_API=3 etcdctl get /kubernetes.io/secrets/default/secret1
```

where [...] contains the additional arguments for connecting to the etcd server.

3. Verify the stored secret is prefixed with k8s:enc:kms:v1: for KMS v1 or prefixed with k8s:enc:kms:v2: for KMS v2, 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
```

The Secret should contain mykey: mydata

Ensuring all secrets are encrypted

When encryption at rest is correctly configured, resources are encrypted on write. Thus we can perform an in-place no-op update to ensure that data is encrypted.

The following command reads all secrets and then updates them to apply server side encryption. If an error occurs due to a conflicting write, retry the command. For larger clusters, you may wish to subdivide the secrets by namespace or script an update.

```
kubectl get secrets --all-namespaces -o json | kubectl replace -f -
```

Switching from a local encryption provider to the KMS provider

To switch from a local encryption provider to the `kms` provider and re-encrypt all of the secrets:

1. Add the `kms` provider as the first entry in the configuration file as shown in the following example.

```
apiVersion: apiserver.config.k8s.io/v1
kind: EncryptionConfiguration
resources:
  - resources:
    - secrets
  providers:
    - kms:
        apiVersion: v2
        name : myKmsPlugin
        endpoint: unix:///tmp/socketfile.sock
    - aescbc:
        keys:
          - name: key1
            secret: <BASE 64 ENCODED SECRET>
```

2. Restart all `kube-apiserver` processes.
3. Run the following command to force all secrets to be re-encrypted using the `kms` provider.

```
kubectl get secrets --all-namespaces -o json | kubectl replace -f -
```

Disabling encryption at rest

To disable encryption at rest:

1. Place the `identity` provider as the first entry in the configuration file:

```
apiVersion: apiserver.config.k8s.io/v1
kind: EncryptionConfiguration
resources:
  - resources:
    - secrets
providers:
  - identity: {}
  - kms:
    apiVersion: v2
    name : myKmsPlugin
    endpoint: unix:///tmp/socketfile.sock
```

2. Restart all `kube-apiserver` processes.

3. Run the following command to force all secrets to be decrypted.

```
kubectl get secrets --all-namespaces -o json | kubectl replace -f
```

2.39 - Using CoreDNS for Service Discovery

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

Before you begin

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

- [Killercoda](#)
- [Play with Kubernetes](#)

Your Kubernetes server must be at or later than version v1.9. To check the version, enter `kubectl version`.

About CoreDNS

[CoreDNS](#) is a flexible, extensible DNS server that can serve as the Kubernetes cluster DNS. Like Kubernetes, the CoreDNS project is hosted by the [CNCF](#).

You can use CoreDNS instead of kube-dns in your cluster by replacing kube-dns in an existing deployment, or by using tools like kubeadm that will deploy and upgrade the cluster for you.

Installing CoreDNS

For manual deployment or replacement of kube-dns, see the documentation at the [CoreDNS GitHub project](#).

Migrating to CoreDNS

Upgrading an existing cluster with kubeadm

In Kubernetes version 1.21, kubeadm removed its support for `kube-dns` as a DNS application. For `kubeadm` v1.28, the only supported cluster DNS application is CoreDNS.

You can move to CoreDNS when you use `kubeadm` to upgrade a cluster that is using `kube-dns`. In this case, `kubeadm` generates the CoreDNS configuration ("Corefile") based upon the `kube-dns` ConfigMap, preserving configurations for stub domains, and upstream name server.

Upgrading CoreDNS

You can check the version of CoreDNS that kubeadm installs for each version of Kubernetes in the page [CoreDNS version in Kubernetes](#).

CoreDNS can be upgraded manually in case you want to only upgrade CoreDNS or use your own custom image. There is a helpful [guideline and walkthrough](#) available to ensure a smooth upgrade. Make sure the existing CoreDNS configuration ("Corefile") is retained when upgrading your cluster.

If you are upgrading your cluster using the `kubeadm` tool, `kubeadm` can take care of retaining the existing CoreDNS configuration automatically.

Tuning CoreDNS

When resource utilisation is a concern, it may be useful to tune the configuration of CoreDNS. For more details, check out the [documentation on scaling CoreDNS](#).

What's next

You can configure [CoreDNS](#) to support many more use cases than kube-dns does by modifying the CoreDNS configuration ("Corefile"). For more information, see the [documentation](#) for the `kubernetes` CoreDNS plugin, or read the [Custom DNS Entries for Kubernetes](#), in the CoreDNS blog.

2.40 - Using NodeLocal DNSCache in Kubernetes Clusters

FEATURE STATE: [Kubernetes v1.18 \[stable\]](#)

This page provides an overview of NodeLocal DNSCache feature in Kubernetes.

Before you begin

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

- [Killercoda](#)
- [Play with Kubernetes](#)

To check the version, enter `kubectl version`.

Introduction

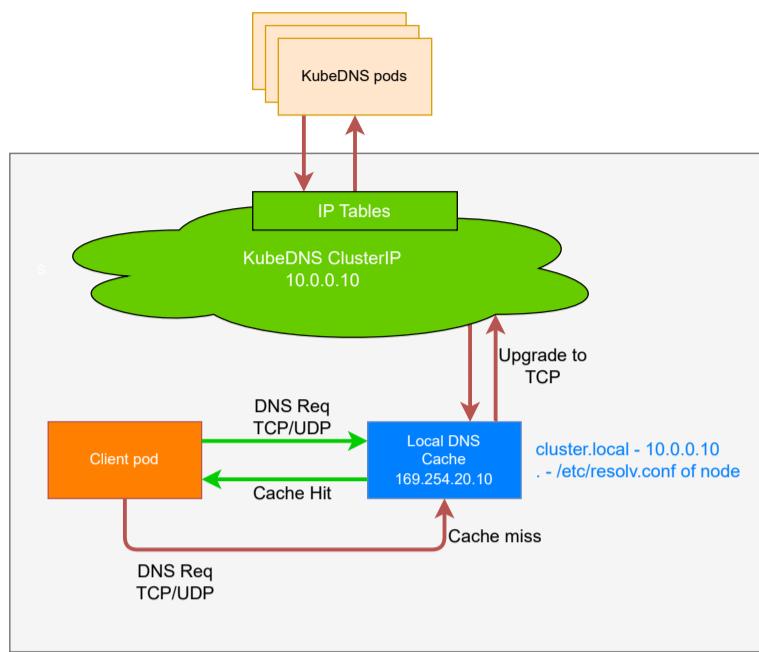
NodeLocal DNSCache improves Cluster DNS performance by running a DNS caching agent on cluster nodes as a DaemonSet. In today's architecture, Pods in 'ClusterFirst' DNS mode reach out to a `kube-dns serviceIP` for DNS queries. This is translated to a `kube-dns/CoreDNS` endpoint via iptables rules added by `kube-proxy`. With this new architecture, Pods will reach out to the DNS caching agent running on the same node, thereby avoiding iptables DNAT rules and connection tracking. The local caching agent will query `kube-dns` service for cache misses of cluster hostnames (" `cluster.local` " suffix by default).

Motivation

- With the current DNS architecture, it is possible that Pods with the highest DNS QPS have to reach out to a different node, if there is no local `kube-dns/CoreDNS` instance. Having a local cache will help improve the latency in such scenarios.
- Skipping iptables DNAT and connection tracking will help reduce [conntrack races](#) and avoid UDP DNS entries filling up conntrack table.
- Connections from the local caching agent to `kube-dns` service can be upgraded to TCP. TCP conntrack entries will be removed on connection close in contrast with UDP entries that have to timeout ([default nf_conntrack_udp_timeout](#) is 30 seconds)
- Upgrading DNS queries from UDP to TCP would reduce tail latency attributed to dropped UDP packets and DNS timeouts usually up to 30s (3 retries + 10s timeout). Since the nodelocal cache listens for UDP DNS queries, applications don't need to be changed.
- Metrics & visibility into DNS requests at a node level.
- Negative caching can be re-enabled, thereby reducing the number of queries for the `kube-dns` service.

Architecture Diagram

This is the path followed by DNS Queries after NodeLocal DNSCache is enabled:



Nodelocal DNSCache flow

This image shows how NodeLocal DNSCache handles DNS queries.

Configuration

Note: The local listen IP address for NodeLocal DNSCache can be any address that can be guaranteed to not collide with any existing IP in your cluster. It's recommended to use an address with a local scope, for example, from the 'link-local' range '169.254.0.0/16' for IPv4 or from the 'Unique Local Address' range in IPv6 'fd00::/8'.

This feature can be enabled using the following steps:

- Prepare a manifest similar to the sample [nodelocaldns.yaml](#) and save it as `nodelocaldns.yaml`.
- If using IPv6, the CoreDNS configuration file needs to enclose all the IPv6 addresses into square brackets if used in 'IP:Port' format. If you are using the sample manifest from the previous point, this will require you to modify [the configuration line L70](#) like this: " `health [__PILLAR__LOCAL__DNS__]:8080` "
- Substitute the variables in the manifest with the right values:

```
kubedns=`kubectl get svc kube-dns -n kube-system -o jsonpath={.spec.ports[?(@.name=="node-local-dns")].nodePort}`  
domain=<cluster-domain>  
localdns=<node-local-address>
```

<cluster-domain> is "cluster.local" by default. <node-local-address> is the local listen IP address chosen for NodeLocal DNSCache.

- If kube-proxy is running in IPTABLES mode:

```
sed -i "s/__PILLAR__LOCAL__DNS__/$localdns/g; s/__PILLAR__DN
```

`__PILLAR__CLUSTER__DNS__` and `__PILLAR__UPSTREAM__SOURCES__` will be populated by the `node-local-dns` pods. In this mode, the `node-local-dns` pods listen on both the kube-dns service IP as well as `<node-local-address>`, so pods can look up DNS records using either IP address.

- o If kube-proxy is running in IPVS mode:

```
sed -i "s/__PILLAR__LOCAL__DNS__/$localdns/g; s/__PILLAR__DN
```

In this mode, the `node-local-dns` pods listen only on `<node-local-address>`. The `node-local-dns` interface cannot bind the kube-dns cluster IP since the interface used for IPVS loadbalancing already uses this address.

`__PILLAR__UPSTREAM__SOURCES__` will be populated by the node-local-dns pods.

- Run `kubectl create -f nodelocaldns.yaml`
- If using kube-proxy in IPVS mode, `--cluster-dns` flag to kubelet needs to be modified to use `<node-local-address>` that NodeLocal DNSCache is listening on. Otherwise, there is no need to modify the value of the `--cluster-dns` flag, since NodeLocal DNSCache listens on both the kube-dns service IP as well as `<node-local-address>`.

Once enabled, the `node-local-dns` Pods will run in the `kube-system` namespace on each of the cluster nodes. This Pod runs [CoreDNS](#) in cache mode, so all CoreDNS metrics exposed by the different plugins will be available on a per-node basis.

You can disable this feature by removing the DaemonSet, using `kubectl delete -f <manifest>`. You should also revert any changes you made to the kubelet configuration.

StubDomains and Upstream server Configuration

StubDomains and upstream servers specified in the `kube-dns` ConfigMap in the `kube-system` namespace are automatically picked up by `node-local-dns` pods. The ConfigMap contents need to follow the format shown in [the example](#). The `node-local-dns` ConfigMap can also be modified directly with the stubDomain configuration in the Corefile format. Some cloud providers might not allow modifying `node-local-dns` ConfigMap directly. In those cases, the `kube-dns` ConfigMap can be updated.

Setting memory limits

The `node-local-dns` Pods use memory for storing cache entries and processing queries. Since they do not watch Kubernetes objects, the cluster size or the number of Services / EndpointSlices do not directly affect memory usage. Memory usage is influenced by the DNS query pattern. From [CoreDNS docs](#),

The default cache size is 10000 entries, which uses about 30 MB when completely filled.

This would be the memory usage for each server block (if the cache gets completely filled). Memory usage can be reduced by specifying smaller cache sizes.

The number of concurrent queries is linked to the memory demand, because each extra goroutine used for handling a query requires an amount of memory. You can set an upper limit using the `max_concurrent` option in the forward plugin.

If a `node-local-dns` Pod attempts to use more memory than is available (because of total system resources, or because of a configured [resource limit](#)), the operating system may shut down that pod's container. If this happens, the container that is terminated ("OOMKilled") does not clean up the custom packet filtering rules that it previously added during startup. The `node-local-dns` container should get restarted (since managed as part of a DaemonSet), but this will lead to a brief DNS downtime each time that the container fails: the packet filtering rules direct DNS queries to a local Pod that is unhealthy.

You can determine a suitable memory limit by running `node-local-dns` pods without a limit and measuring the peak usage. You can also set up and use a [VerticalPodAutoscaler](#) in *recommender mode*, and then check its recommendations.

2.41 - Using sysctls in a Kubernetes Cluster

FEATURE STATE: [Kubernetes v1.21 \[stable\]](#)

This document describes how to configure and use kernel parameters within a Kubernetes cluster using the [sysctl](#) interface.

Note: Starting from Kubernetes version 1.23, the kubelet supports the use of either `/` or `.` as separators for sysctl names. Starting from Kubernetes version 1.25, setting Sysctls for a Pod supports setting sysctls with slashes. For example, you can represent the same sysctl name as `kernel.shm_rmid_forced` using a period as the separator, or as `kernel/shm_rmid_forced` using a slash as a separator. For more sysctl parameter conversion method details, please refer to the page [sysctl.d\(5\)](#) from the Linux man-pages project.

Before you begin

Note: `sysctl` is a Linux-specific command-line tool used to configure various kernel parameters and it is not available on non-Linux operating systems.

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

- [KillerCoda](#)
- [Play with Kubernetes](#)

For some steps, you also need to be able to reconfigure the command line options for the kubelets running on your cluster.

Listing all Sysctl Parameters

In Linux, the sysctl interface allows an administrator to modify kernel parameters at runtime. Parameters are available via the `/proc/sys/` virtual process file system. The parameters cover various subsystems such as:

- kernel (common prefix: `kernel.`)
- networking (common prefix: `net.`)
- virtual memory (common prefix: `vm.`)
- MDADM (common prefix: `dev.`)
- More subsystems are described in [Kernel docs](#).

To get a list of all parameters, you can run

```
sudo sysctl -a
```

Safe and Unsafe Sysctls

Kubernetes classes sysctls as either *safe* or *unsafe*. In addition to proper namespacing, a *safe* sysctl must be properly *isolated* between pods on the same node. This means that setting a *safe* sysctl for one pod

- must not have any influence on any other pod on the node
- must not allow to harm the node's health
- must not allow to gain CPU or memory resources outside of the resource limits of a pod.

By far, most of the *namespaced* sysctls are not necessarily considered *safe*.

The following sysctls are supported in the *safe* set:

- `kernel.shm_rmid_forced`,
- `net.ipv4.ip_local_port_range`,
- `net.ipv4.tcp_syncookies`,
- `net.ipv4.ping_group_range` (since Kubernetes 1.18),
- `net.ipv4.ip_unprivileged_port_start` (since Kubernetes 1.22).

Note:

There are some exceptions to the set of safe sysctls:

- The `net.*` sysctls are not allowed with host networking enabled.
- The `net.ipv4.tcp_syncookies` sysctl 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.

Enabling Unsafe Sysctls

All *safe* sysctls are enabled by default.

All *unsafe* sysctls are disabled by default and must be allowed manually by the cluster admin on a per-node basis. Pods with disabled unsafe sysctls will be scheduled, but will fail to launch.

With the warning above in mind, the cluster admin can allow certain *unsafe* sysctls for very special situations such as high-performance or real-time application tuning. *Unsafe* sysctls are enabled on a node-by-node basis with a flag of the kubelet; for example:

```
kubelet --allowed-unsafe-sysctls \
'kernel.msg*,net.core.somaxconn' ...
```

For Minikube, this can be done via the `extra-config` flag:

```
minikube start --extra-config="kubelet.allowed-unsafe-sysctls=kernel.m
```

Only *namespaced* sysctls can be enabled this way.

Setting Sysctls for a Pod

A number of sysctls are *namespaced* in today's Linux kernels. This means that they can be set independently for each pod on a node. Only namespaced sysctls are configurable via the pod securityContext within Kubernetes.

The following sysctls are known to be namespaced. This list could change in future versions of the Linux kernel.

- `kernel.shm*`,
- `kernel.msg*`,
- `kernel.sem`,
- `fs.mqueue.*`,
- Those `net.*` that can be set in container networking namespace. However, there are exceptions (e.g., `net.netfilter.nf_conntrack_max` and `net.netfilter.nf_conntrack_expect_max` can be set in container networking namespace but are unnamespaced before Linux 5.12.2).

Sysctls with no namespace are called *node-level* sysctls. If you need to set them, you must manually configure them on each node's operating system, or by using a DaemonSet with privileged containers.

Use the pod securityContext to configure namespaced sysctls. The securityContext applies to all containers in the same pod.

This example uses the pod securityContext to set a safe sysctl `kernel.shm_rmid_forced` and two unsafe sysctls `net.core.somaxconn` and `kernel.msgmax`. There is no distinction between *safe* and *unsafe* sysctls in the specification.

Warning: Only modify sysctl parameters after you understand their effects, to avoid destabilizing your operating system.

```
apiVersion: v1
kind: Pod
metadata:
  name: sysctl-example
spec:
  securityContext:
    sysctls:
      - name: kernel.shm_rmid_forced
        value: "0"
      - name: net.core.somaxconn
        value: "1024"
      - name: kernel.msgmax
        value: "65536"
    ...
  
```

Warning: Due to their nature of being *unsafe*, the use of *unsafe* sysctls is at-your-own-risk and can lead to severe problems like wrong behavior of containers, resource shortage or complete breakage of a node.

It is good practice to consider nodes with special sysctl settings as *tainted* within a cluster, and only schedule pods onto them which need those sysctl settings. It is suggested to use the Kubernetes [taints and toleration feature](#) to implement this.

A pod with the *unsafe* sysctls will fail to launch on any node which has not enabled those two *unsafe* sysctls explicitly. As with *node-level* sysctls it is recommended to use [*taints and toleration feature*](#) or [*taints on nodes*](#) to schedule those pods onto the right nodes.

2.42 - Utilizing the NUMA-aware Memory Manager

FEATURE STATE: [Kubernetes v1.22 \[beta\]](#)

The Kubernetes *Memory Manager* enables the feature of guaranteed memory (and hugepages) allocation for pods in the `Guaranteed` QoS class.

The Memory Manager employs hint generation protocol to yield the most suitable NUMA affinity for a pod. The Memory Manager feeds the central manager (*Topology Manager*) with these affinity hints. Based on both the hints and Topology Manager policy, the pod is rejected or admitted to the node.

Moreover, the Memory Manager ensures that the memory which a pod requests is allocated from a minimum number of NUMA nodes.

The Memory Manager is only pertinent to Linux based hosts.

Before you begin

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

- [Killercoda](#)
- [Play with Kubernetes](#)

Your Kubernetes server must be at or later than version v1.21. To check the version, enter `kubectl version`.

To align memory resources with other requested resources in a Pod spec:

- the CPU Manager should be enabled and proper CPU Manager policy should be configured on a Node. See [control CPU Management Policies](#);
- the Topology Manager should be enabled and proper Topology Manager policy should be configured on a Node. See [control Topology Management Policies](#).

Starting from v1.22, the Memory Manager is enabled by default through `MemoryManager` [feature gate](#).

Preceding v1.22, the `kubelet` must be started with the following flag:

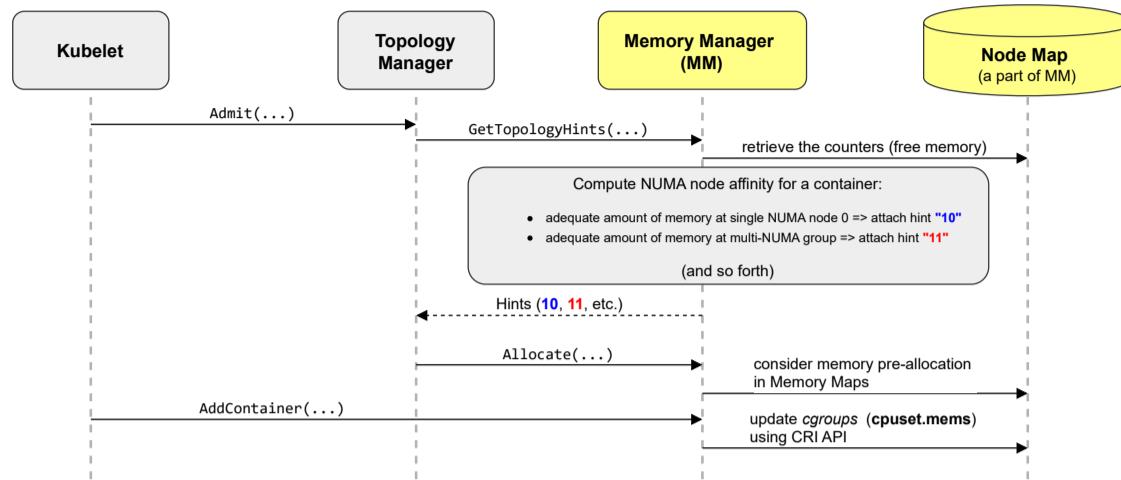
```
--feature-gates=MemoryManager=true
```

in order to enable the Memory Manager feature.

How Memory Manager Operates?

The Memory Manager currently offers the guaranteed memory (and hugepages) allocation for Pods in Guaranteed QoS class. To immediately put the Memory Manager into operation follow the guidelines in the section [Memory Manager configuration](#), and subsequently, prepare and deploy a `Guaranteed` pod as illustrated in the section [Placing a Pod in the Guaranteed QoS class](#).

The Memory Manager is a Hint Provider, and it provides topology hints for the Topology Manager which then aligns the requested resources according to these topology hints. It also enforces cgroups (i.e. `cputset.mems`) for pods. The complete flow diagram concerning pod admission and deployment process is illustrated in [Memory Manager KEP: Design Overview](#) and below:



During this process, the Memory Manager updates its internal counters stored in [Node Map and Memory Maps](#) to manage guaranteed memory allocation.

The Memory Manager updates the Node Map during the startup and runtime as follows.

Startup

This occurs once a node administrator employs `--reserved-memory` (section [Reserved memory flag](#)). In this case, the Node Map becomes updated to reflect this reservation as illustrated in [Memory Manager KEP: Memory Maps at start-up \(with examples\)](#).

The administrator must provide `--reserved-memory` flag when `Static` policy is configured.

Runtime

Reference [Memory Manager KEP: Memory Maps at runtime \(with examples\)](#) illustrates how a successful pod deployment affects the Node Map, and it also relates to how potential Out-of-Memory (OOM) situations are handled further by Kubernetes or operating system.

Important topic in the context of Memory Manager operation is the management of NUMA groups. Each time pod's memory request is in excess of single NUMA node capacity, the Memory Manager attempts to create a group that comprises several NUMA nodes and features extend memory capacity. The problem has been solved as elaborated in [Memory Manager KEP: How to enable the guaranteed memory allocation over many NUMA nodes?](#). Also, reference [Memory Manager KEP: Simulation - how the Memory Manager works? \(by examples\)](#) illustrates how the management of groups occurs.

Memory Manager configuration

Other Managers should be first pre-configured. Next, the Memory Manager feature should be enabled and be run with `Static` policy (section [Static policy](#)). Optionally, some amount of memory can be reserved for system or kubelet processes to increase node stability (section [Reserved memory flag](#)).

Policies

Memory Manager supports two policies. You can select a policy via a

kubelet flag `--memory-manager-policy` :

- None (default)
- Static

None policy

This is the default policy and does not affect the memory allocation in any way. It acts the same as if the Memory Manager is not present at all.

The `None` policy returns default topology hint. This special hint denotes that Hint Provider (Memory Manager in this case) has no preference for NUMA affinity with any resource.

Static policy

In the case of the `Guaranteed` pod, the `Static` Memory Manager policy returns topology hints relating to the set of NUMA nodes where the memory can be guaranteed, and reserves the memory through updating the internal [NodeMap](#) object.

In the case of the `BestEffort` or `Burstable` pod, the `Static` Memory Manager policy sends back the default topology hint as there is no request for the guaranteed memory, and does not reserve the memory in the internal [NodeMap](#) object.

Reserved memory flag

The [Node Allocatable](#) mechanism is commonly used by node administrators to reserve K8S node system resources for the kubelet or operating system processes in order to enhance the node stability. A dedicated set of flags can be used for this purpose to set the total amount of reserved memory for a node. This pre-configured value is subsequently utilized to calculate the real amount of node's "allocatable" memory available to pods.

The Kubernetes scheduler incorporates "allocatable" to optimise pod scheduling process. The foregoing flags include `--kube-reserved`, `--system-reserved` and `--eviction-threshold`. The sum of their values will account for the total amount of reserved memory.

A new `--reserved-memory` flag was added to Memory Manager to allow for this total reserved memory to be split (by a node administrator) and accordingly reserved across many NUMA nodes.

The flag specifies a comma-separated list of memory reservations of different memory types per NUMA node. Memory reservations across multiple NUMA nodes can be specified using semicolon as separator. This parameter is only useful in the context of the Memory Manager feature. The Memory Manager will not use this reserved memory for the allocation of container workloads.

For example, if you have a NUMA node "NUMA0" with `10Gi` of memory available, and the `--reserved-memory` was specified to reserve `1Gi` of memory at "NUMA0", the Memory Manager assumes that only `9Gi` is available for containers.

You can omit this parameter, however, you should be aware that the quantity of reserved memory from all NUMA nodes should be equal to the quantity of memory specified by the [Node Allocatable feature](#). If at least one node allocatable parameter is non-zero, you will need to specify `--`

`--reserved-memory` for at least one NUMA node. In fact, `eviction-hard` threshold value is equal to `100Mi` by default, so if `Static` policy is used, `--reserved-memory` is obligatory.

Also, avoid the following configurations:

1. duplicates, i.e. the same NUMA node or memory type, but with a different value;
2. setting zero limit for any of memory types;
3. NUMA node IDs that do not exist in the machine hardware;
4. memory type names different than `memory` OR `hugepages-<size>` (hugepages of particular `<size>` should also exist).

Syntax:

```
--reserved-memory N:memory-type1=value1,memory-type2=value2,...
```

- `N` (integer) - NUMA node index, e.g. `0`
- `memory-type` (string) - represents memory type:
 - `memory` - conventional memory
 - `hugepages-2Mi` OR `hugepages-1Gi` - hugepages
- `value` (string) - the quantity of reserved memory, e.g. `1Gi`

Example usage:

```
--reserved-memory 0:memory=1Gi,hugepages-1Gi=2Gi
```

or

```
--reserved-memory 0:memory=1Gi --reserved-memory 1:memory=2Gi
```

or

```
--reserved-memory '0:memory=1Gi;1:memory=2Gi'
```

When you specify values for `--reserved-memory` flag, you must comply with the setting that you prior provided via Node Allocatable Feature flags. That is, the following rule must be obeyed for each memory type:

```
sum(reserved-memory(i)) = kube-reserved + system-reserved + eviction-threshold ,
```

where `i` is an index of a NUMA node.

If you do not follow the formula above, the Memory Manager will show an error on startup.

In other words, the example above illustrates that for the conventional memory (`type=memory`), we reserve `3Gi` in total, i.e.:

```
sum(reserved-memory(i)) = reserved-memory(0) + reserved-memory(1) = 1Gi + 2Gi = 3Gi
```

An example of kubelet command-line arguments relevant to the node Allocatable configuration:

- `--kube-reserved=cpu=500m,memory=50Mi`
- `--system-reserved=cpu=123m,memory=333Mi`
- `--eviction-hard=memory.available<500Mi`

Note: The default hard eviction threshold is `100MiB`, and **not** zero. Remember to increase the quantity of memory that you reserve by setting `--reserved-memory` by that hard eviction threshold. Otherwise, the kubelet will not start Memory Manager and display an error.

Here is an example of a correct configuration:

```
--feature-gates=MemoryManager=true
--kube-reserved=cpu=4,memory=4Gi
--system-reserved=cpu=1,memory=1Gi
--memory-manager-policy=Static
--reserved-memory '0:memory=3Gi;1:memory=2148Mi'
```

Let us validate the configuration above:

1. kube-reserved + system-reserved + eviction-hard(default) = reserved-memory(0) + reserved-memory(1)
2. 4GiB + 1GiB + 100MiB = 3GiB + 2148MiB
3. 5120MiB + 100MiB = 3072MiB + 2148MiB
4. 5220MiB = 5220MiB (which is correct)

Placing a Pod in the Guaranteed QoS class

If the selected policy is anything other than `None`, the Memory Manager identifies pods that are in the `Guaranteed` QoS class. The Memory Manager provides specific topology hints to the Topology Manager for each `Guaranteed` pod. For pods in a QoS class other than `Guaranteed`, the Memory Manager provides default topology hints to the Topology Manager.

The following excerpts from pod manifests assign a pod to the `Guaranteed` QoS class.

Pod with integer CPU(s) runs in the `Guaranteed` QoS class, when `requests` are equal to `limits`:

```
spec:
  containers:
  - name: nginx
    image: nginx
    resources:
      limits:
        memory: "200Mi"
        cpu: "2"
        example.com/device: "1"
      requests:
        memory: "200Mi"
        cpu: "2"
        example.com/device: "1"
```

Also, a pod sharing CPU(s) runs in the `Guaranteed` QoS class, when `requests` are equal to `limits`.

```

spec:
  containers:
    - name: nginx
      image: nginx
      resources:
        limits:
          memory: "200Mi"
          cpu: "300m"
          example.com/device: "1"
        requests:
          memory: "200Mi"
          cpu: "300m"
          example.com/device: "1"

```

Notice that both CPU and memory requests must be specified for a Pod to lend it to Guaranteed QoS class.

Troubleshooting

The following means can be used to troubleshoot the reason why a pod could not be deployed or became rejected at a node:

- pod status - indicates topology affinity errors
- system logs - include valuable information for debugging, e.g., about generated hints
- state file - the dump of internal state of the Memory Manager (includes [Node Map and Memory Maps](#))
- starting from v1.22, the [device plugin resource API](#) can be used to retrieve information about the memory reserved for containers

Pod status (TopologyAffinityError)

This error typically occurs in the following situations:

- a node has not enough resources available to satisfy the pod's request
- the pod's request is rejected due to particular Topology Manager policy constraints

The error appears in the status of a pod:

```
kubectl get pods
```

NAME	READY	STATUS	RESTARTS	AGE
guaranteed	0/1	TopologyAffinityError	0	113s

Use `kubectl describe pod <id>` or `kubectl get events` to obtain detailed error message:

```
Warning  TopologyAffinityError  10m   kubelet, dell18  Resources cannot
```

System logs

Search system logs with respect to a particular pod.

The set of hints that Memory Manager generated for the pod can be found in the logs. Also, the set of hints generated by CPU Manager should be present in the logs.

Topology Manager merges these hints to calculate a single best hint. The best hint should be also present in the logs.

The best hint indicates where to allocate all the resources. Topology Manager tests this hint against its current policy, and based on the verdict, it either admits the pod to the node or rejects it.

Also, search the logs for occurrences associated with the Memory Manager, e.g. to find out information about `cgroups` and `cpuset.mems` updates.

Examine the memory manager state on a node

Let us first deploy a sample `Guaranteed` pod whose specification is as follows:

```
apiVersion: v1
kind: Pod
metadata:
  name: guaranteed
spec:
  containers:
  - name: guaranteed
    image: consumer
    imagePullPolicy: Never
    resources:
      limits:
        cpu: "2"
        memory: 150Gi
      requests:
        cpu: "2"
        memory: 150Gi
    command: ["sleep", "infinity"]
```

Next, let us log into the node where it was deployed and examine the state file in `/var/lib/kubelet/memory_manager_state`:

```
{  
    "policyName": "Static",  
    "machineState": {  
        "0": {  
            "numberOfAssignments": 1,  
            "memoryMap": {  
                "hugepages-1Gi": {  
                    "total": 0,  
                    "systemReserved": 0,  
                    "allocatable": 0,  
                    "reserved": 0,  
                    "free": 0  
                },  
                "memory": {  
                    "total": 134987354112,  
                    "systemReserved": 3221225472,  
                    "allocatable": 131766128640,  
                    "reserved": 131766128640,  
                    "free": 0  
                }  
            },  
            "nodes": [  
                0,  
                1  
            ]  
        },  
        "1": {  
            "numberOfAssignments": 1,  
            "memoryMap": {  
                "hugepages-1Gi": {  
                    "total": 0,  
                    "systemReserved": 0,  
                    "allocatable": 0,  
                    "reserved": 0,  
                    "free": 0  
                },  
                "memory": {  
                    "total": 135286722560,  
                    "systemReserved": 2252341248,  
                    "allocatable": 133034381312,  
                    "reserved": 29295144960,  
                    "free": 103739236352  
                }  
            },  
            "nodes": [  
                0,  
                1  
            ]  
        }  
    },  
    "entries": {  
        "fa9bdd38-6df9-4cf9-aa67-8c4814da37a8": {  
            "guaranteed": [  
                {  
                    "numaAffinity": [  
                        0,  
                        1  
                    ],  
                    "type": "memory",  
                    "size": 161061273600  
                }  
            ]  
        },  
        "checksum": 4142013182  
    }  
}
```

It can be deduced from the state file that the pod was pinned to both NUMA nodes, i.e.:

```
"numaAffinity":  
  0,  
  1  
,
```

Pinned term means that pod's memory consumption is constrained (through `cgroups` configuration) to these NUMA nodes.

This automatically implies that Memory Manager instantiated a new group that comprises these two NUMA nodes, i.e. `0` and `1` indexed NUMA nodes.

Notice that the management of groups is handled in a relatively complex manner, and further elaboration is provided in Memory Manager KEP in [this](#) and [this](#) sections.

In order to analyse memory resources available in a group, the corresponding entries from NUMA nodes belonging to the group must be added up.

For example, the total amount of free "conventional" memory in the group can be computed by adding up the free memory available at every NUMA node in the group, i.e., in the `"memory"` section of NUMA node `0` (`"free":0`) and NUMA node `1` (`"free":103739236352`). So, the total amount of free "conventional" memory in this group is equal to `0 + 103739236352` bytes.

The line `"systemReserved":3221225472` indicates that the administrator of this node reserved `3221225472` bytes (i.e. `3Gi`) to serve kubelet and system processes at NUMA node `0`, by using `--reserved-memory` flag.

Device plugin resource API

The kubelet provides a `PodResourceLister` gRPC service to enable discovery of resources and associated metadata. By using its [List gRPC endpoint](#), information about reserved memory for each container can be retrieved, which is contained in protobuf `ContainerMemory` message. This information can be retrieved solely for pods in Guaranteed QoS class.

What's next

- [Memory Manager KEP: Design Overview](#)
- [Memory Manager KEP: Memory Maps at start-up \(with examples\)](#)
- [Memory Manager KEP: Memory Maps at runtime \(with examples\)](#)
- [Memory Manager KEP: Simulation - how the Memory Manager works? \(by examples\)](#)
- [Memory Manager KEP: The Concept of Node Map and Memory Maps](#)
- [Memory Manager KEP: How to enable the guaranteed memory allocation over many NUMA nodes?](#)

2.43 - Verify Signed Kubernetes Artifacts

FEATURE STATE: [Kubernetes v1.26 \[beta\]](#)

Before you begin

You will need to have the following tools installed:

- `cosign` ([install guide](#))
- `curl` (often provided by your operating system)
- `jq` ([download jq](#))

Verifying binary signatures

The Kubernetes release process signs all binary artifacts (tarballs, SPDX files, standalone binaries) by using cosign's keyless signing. To verify a particular binary, retrieve it together with its signature and certificate:

```
URL=https://dl.k8s.io/release/v1.28.3/bin/linux/amd64
BINARY=kubectl

FILES=(
    "$BINARY"
    "$BINARY.sig"
    "$BINARY.cert"
)

for FILE in "${FILES[@]}"; do
    curl -sSfL --retry 3 --retry-delay 3 "$URL/$FILE" -o "$FILE"
done
```

Then verify the blob by using `cosign verify-blob`:

```
cosign verify-blob "$BINARY" \
--signature "$BINARY".sig \
--certificate "$BINARY".cert \
--certificate-identity krel-staging@k8s-releng-prod.iam.gserviceaccount.com \
--certificate-oidc-issuer https://accounts.google.com
```

Note:

Cosign 2.0 requires the `--certificate-identity` and `--certificate-oidc-issuer` options.

To learn more about keyless signing, please refer to [Keyless Signatures](#).

Previous versions of Cosign required that you set
`COSIGN_EXPERIMENTAL=1`.

For additional information, please refer to the [sigstore Blog](#)

Verifying image signatures

For a complete list of images that are signed please refer to [Releases](#).

Pick one image from this list and verify its signature using the `cosign verify` command:

```
cosign verify registry.k8s.io/kube-apiserver-amd64:v1.28.3 \
--certificate-identity krel-trust@k8s-releng-prod.iam.gserviceaccount.com \
--certificate-oidc-issuer https://accounts.google.com \
| jq .
```

Verifying images for all control plane components

To verify all signed control plane images for the latest stable version (v1.28.3), please run the following commands:

```
curl -Ls "https://sbom.k8s.io/$(curl -Ls https://dl.k8s.io/release/stable \
| grep "SPDXID: SPDXRef-Package-registry.k8s.io" \
| grep -v sha256 | cut -d- -f3- | sed 's/-/\//' | sed 's/-v1/:v1/' \
| sort > images.txt

```

Once you have verified an image, you can specify the image by its digest in your Pod manifests as per this example:

```
registry-url/image-name@sha256:45b23dee08af5e43a7fea6c4cf9c25ccf269ee1:
```

For more information, please refer to the [Image Pull Policy](#) section.

Verifying Image Signatures with Admission Controller

For non-control plane images (for example [conformance image](#)), signatures can also be verified at deploy time using [sigstore policy-controller](#) admission controller.

Here are some helpful resources to get started with `policy-controller`:

- [Installation](#)
- [Configuration Options](#)

Verify the Software Bill Of Materials

You can verify the Kubernetes Software Bill of Materials (SBOM) by using the sigstore certificate and signature, or the corresponding SHA files:

```
# Retrieve the latest available Kubernetes release version
VERSION=$(curl -Ls https://dl.k8s.io/release/stable.txt)

# Verify the SHA512 sum
curl -Ls "https://sbom.k8s.io/$VERSION/release" -o "$VERSION.spdx"
echo "$(curl -Ls "https://sbom.k8s.io/$VERSION/release.sha512")" $VERSION.spdx

# Verify the SHA256 sum
echo "$(curl -Ls "https://sbom.k8s.io/$VERSION/release.sha256")" $VERSION.spdx

# Retrieve sigstore signature and certificate
curl -Ls "https://sbom.k8s.io/$VERSION/release.sig" -o "$VERSION.spdx.sig"
curl -Ls "https://sbom.k8s.io/$VERSION/release.cert" -o "$VERSION.spdx.cert"

# Verify the sigstore signature
cosign verify-blob \
  --certificate "$VERSION.spdx.cert" \
  --signature "$VERSION.spdx.sig" \
  --certificate-identity krel-staging@k8s-releng-prod.iam.gserviceaccount.com \
  --certificate-oidc-issuer https://accounts.google.com \
  "$VERSION.spdx"
```

3 - Configure Pods and Containers

Perform common configuration tasks for Pods and containers.

3.1 - 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

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

- [Killercoda](#)
- [Play with Kubernetes](#)

To check the version, enter `kubectl version`.

Each node in your cluster must have at least 300 MiB of memory.

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

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

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

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

```
kubectl get apiservices
```

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

NAME
v1beta1.metrics.k8s.io

Create a namespace

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

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

Specify a memory request and a memory limit

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

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

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

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

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

Create the Pod:

```
kubectl apply -f https://k8s.io/examples/pods/resource/memory-request-;
```

Verify that the Pod Container is running:

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

View detailed information about the Pod:

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

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

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

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

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

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

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

Delete your Pod:

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

Exceed a Container's memory limit

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

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

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

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

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

Create the Pod:

```
kubectl apply -f https://k8s.io/examples/pods/resource/memory-request-1
```

View detailed information about the Pod:

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

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

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

Get a more detailed view of the Container status:

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

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

```
lastState:  
  terminated:  
    containerID: 65183c1877aaec2e8427bc95609cc52677a454b56fc24340dbd:  
    exitCode: 137  
    finishedAt: 2017-06-20T20:52:19Z  
    reason: OOMKilled  
    startedAt: null
```

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

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

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

NAME	READY	STATUS	RESTARTS	AGE
memory-demo-2	0/1	OOMKilled	1	37s

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

View detailed information about the Pod history:

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

The output shows that the Container starts and fails repeatedly:

```
... Normal  Created  Created container with id 66a3a20aa7980e61be49227
... 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 (stre
```

Delete your Pod:

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

Specify a memory request that is too big for your Nodes

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

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

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

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

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

Create the Pod:

```
kubectl apply -f https://k8s.io/examples/pods/resource/memory-request-1.yaml
```

View the Pod status:

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

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

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

View detailed information about the Pod, including events:

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

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

```
Events:
  ...
  Reason           Message
  -----          -----
  ... FailedScheduling  No nodes are available that match all of the t
```

Memory units

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

approximately the same value:

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

Delete your Pod:

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

If you do not specify a memory limit

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

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

Motivation for memory requests and limits

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

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

Clean up

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

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

What's next

For app developers

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

For cluster administrators

- [Configure Default Memory Requests and Limits for a Namespace](#)
- [Configure Default CPU Requests and Limits for a Namespace](#)
- [Configure Minimum and Maximum Memory Constraints for a Namespace](#)
- [Configure Minimum and Maximum CPU Constraints for a Namespace](#)
- [Configure Memory and CPU Quotas for a Namespace](#)
- [Configure a Pod Quota for a Namespace](#)
- [Configure Quotas for API Objects](#)
- [Resize CPU and Memory Resources assigned to Containers](#)

3.2 - Assign CPU Resources to Containers and Pods

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

Before you begin

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

- [KillerCoda](#)
- [Play with Kubernetes](#)

To check the version, enter `kubectl version`.

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

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

If you are running [Minikube](#), run the following command to enable metrics-server:

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

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

```
kubectl get apiservices
```

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

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

Create a namespace

Create a [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 resource manifest. To specify a CPU limit, include `resources:limits`.

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

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

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

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

Create the Pod:

```
kubectl apply -f https://k8s.io/examples/pods/resource/cpu-request-lim:
```

Verify that the Pod is running:

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

View detailed information about the Pod:

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

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

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

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

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

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

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

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

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

CPU units

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

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

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

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

Delete your Pod:

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

Specify a CPU request that is too big for your Nodes

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

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

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

```
pods/resource/cpu-request-limit-2.yaml ┌─────────┐
```

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

Create the Pod:

```
kubectl apply -f https://k8s.io/examples/pods/resource/cpu-request-lim: ┌─────────┐
```

View the Pod status:

```
kubectl get pod cpu-demo-2 --namespace=cpu-example ┌─────────┐
```

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

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: [No��满CPU]

Delete your Pod:

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

If you do not specify a CPU limit

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

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

If you specify a CPU limit but do not specify a CPU request

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

Motivation for CPU requests and limits

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

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

Clean up

Delete your namespace:

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

What's next

For app developers

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

For cluster administrators

- [Configure Default Memory Requests and Limits for a Namespace](#)
- [Configure Default CPU Requests and Limits for a Namespace](#)
- [Configure Minimum and Maximum Memory Constraints for a Namespace](#)
- [Configure Minimum and Maximum CPU Constraints for a Namespace](#)
- [Configure Memory and CPU Quotas for a Namespace](#)
- [Configure a Pod Quota for a Namespace](#)
- [Configure Quotas for API Objects](#)
- [Resize CPU and Memory Resources assigned to Containers](#)

3.3 - Configure GMSA for Windows Pods and containers

FEATURE STATE: [Kubernetes v1.18 \[stable\]](#)

This page shows how to configure [Group Managed Service Accounts](#) (GMSA) for Pods and containers that will run on Windows nodes. Group Managed Service Accounts are a specific type of Active Directory account that provides automatic password management, simplified service principal name (SPN) management, and the ability to delegate the management to other administrators across multiple servers.

In Kubernetes, GMSA credential specs are configured at a Kubernetes cluster-wide scope as Custom Resources. Windows Pods, as well as individual containers within a Pod, can be configured to use a GMSA for domain based functions (e.g. Kerberos authentication) when interacting with other Windows services.

Before you begin

You need to have a Kubernetes cluster and the `kubectl` command-line tool must be configured to communicate with your cluster. The cluster is expected to have Windows worker nodes. This section covers a set of initial steps required once for each cluster:

Install the GMSACredentialSpec CRD

A [CustomResourceDefinition](#)(CRD) for GMSA credential spec resources needs to be configured on the cluster to define the custom resource type `GMSACredentialSpec`. Download the GMSA CRD [YAML](#) and save it as `gmsa-crd.yaml`. Next, install the CRD with `kubectl apply -f gmsa-crd.yaml`

Install webhooks to validate GMSA users

Two webhooks need to be configured on the Kubernetes cluster to populate and validate GMSA credential spec references at the Pod or container level:

1. A mutating webhook that expands references to GMSAs (by name from a Pod specification) into the full credential spec in JSON form within the Pod spec.
2. A validating webhook ensures all references to GMSAs are authorized to be used by the Pod service account.

Installing the above webhooks and associated objects require the steps below:

1. Create a certificate key pair (that will be used to allow the webhook container to communicate to the cluster)
2. Install a secret with the certificate from above.
3. Create a deployment for the core webhook logic.
4. Create the validating and mutating webhook configurations referring to the deployment.

A [script](#) can be used to deploy and configure the GMSA webhooks and associated objects mentioned above. The script can be run with a `--dry-run=server` option to allow you to review the changes that would be made

to your cluster.

The [YAML template](#) used by the script may also be used to deploy the webhooks and associated objects manually (with appropriate substitutions for the parameters)

Configure GMSAs and Windows nodes in Active Directory

Before Pods in Kubernetes can be configured to use GMSAs, the desired GMSAs need to be provisioned in Active Directory as described in the [Windows GMSA documentation](#). Windows worker nodes (that are part of the Kubernetes cluster) need to be configured in Active Directory to access the secret credentials associated with the desired GMSA as described in the [Windows GMSA documentation](#).

Create GMSA credential spec resources

With the GMSACredentialSpec CRD installed (as described earlier), custom resources containing GMSA credential specs can be configured. The GMSA credential spec does not contain secret or sensitive data. It is information that a container runtime can use to describe the desired GMSA of a container to Windows. GMSA credential specs can be generated in YAML format with a utility [PowerShell script](#).

Following are the steps for generating a GMSA credential spec YAML manually in JSON format and then converting it:

1. Import the CredentialSpec [module](#): `ipmo CredentialSpec.psm1`
2. Create a credential spec in JSON format using `New-CredentialSpec`. To create a GMSA credential spec named WebApp1, invoke `New-CredentialSpec -Name WebApp1 -AccountName WebApp1 -Domain $(Get-ADDomain -Current LocalComputer)`
3. Use `Get-CredentialSpec` to show the path of the JSON file.
4. Convert the credspec file from JSON to YAML format and apply the necessary header fields `apiVersion`, `kind`, `metadata` and `credspec` to make it a GMSACredentialSpec custom resource that can be configured in Kubernetes.

The following YAML configuration describes a GMSA credential spec named `gmsa-WebApp1`:

```

apiVersion: windows.k8s.io/v1
kind: GMSACredentialSpec
metadata:
  name: gmsa-WebApp1 # This is an arbitrary name but it will be used to identify the credential spec
credspec:
  ActiveDirectoryConfig:
    GroupManagedServiceAccounts:
      - Name: WebApp1 # Username of the GMSA account
        Scope: CONTOSO # NETBIOS Domain Name
      - Name: WebApp1 # Username of the GMSA account
        Scope: contoso.com # DNS Domain Name
  CmsPlugins:
    - ActiveDirectory
  DomainJoinConfig:
    DnsName: contoso.com # DNS Domain Name
    DnsTreeName: contoso.com # DNS Domain Name Root
    Guid: 244818ae-87ac-4fcd-92ec-e79e5252348a # GUID
    MachineAccountName: WebApp1 # Username of the GMSA account
    NetBiosName: CONTOSO # NETBIOS Domain Name
    Sid: S-1-5-21-2126449477-2524075714-3094792973 # SID of GMSA

```

The above credential spec resource may be saved as `gmsa-Webapp1-credspec.yaml` and applied to the cluster using: `kubectl apply -f gmsa-Webapp1-credspec.yaml`

Configure cluster role to enable RBAC on specific GMSA credential specs

A cluster role needs to be defined for each GMSA credential spec resource. This authorizes the `use` verb on a specific GMSA resource by a subject which is typically a service account. The following example shows a cluster role that authorizes usage of the `gmsa-WebApp1` credential spec from above. Save the file as `gmsa-webapp1-role.yaml` and apply using `kubectl apply -f gmsa-webapp1-role.yaml`

```

# Create the Role to read the credspec
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRole
metadata:
  name: webapp1-role
rules:
  - apiGroups: ["windows.k8s.io"]
    resources: ["gmsacredentialspecs"]
    verbs: ["use"]
    resourceNames: ["gmsa-WebApp1"]

```

Assign role to service accounts to use specific GMSA credsspecs

A service account (that Pods will be configured with) needs to be bound to the cluster role created above. This authorizes the service account to use the desired GMSA credential spec resource. The following shows the default service account being bound to a cluster role `webapp1-role` to use `gmsa-WebApp1` credential spec resource created above.

```
apiVersion: rbac.authorization.k8s.io/v1
kind: RoleBinding
metadata:
  name: allow-default-svc-account-read-on-gmsa-WebApp1
  namespace: default
subjects:
- kind: ServiceAccount
  name: default
  namespace: default
roleRef:
  kind: ClusterRole
  name: webapp1-role
  apiGroup: rbac.authorization.k8s.io
```

Configure GMSA credential spec reference in Pod spec

The Pod spec field `securityContext.windowsOptions.gmsaCredentialSpecName` is used to specify references to desired GMSA credential spec custom resources in Pod specs. This configures all containers in the Pod spec to use the specified GMSA. A sample Pod spec with the annotation populated to refer to `gmsa-WebApp1` :

```
apiVersion: apps/v1
kind: Deployment
metadata:
  labels:
    run: with-creds
  name: with-creds
  namespace: default
spec:
  replicas: 1
  selector:
    matchLabels:
      run: with-creds
  template:
    metadata:
      labels:
        run: with-creds
    spec:
      securityContext:
        windowsOptions:
          gmsaCredentialSpecName: gmsa-webapp1
      containers:
        - image: mcr.microsoft.com/windows/servercore/iis:windowsservercore-ltsc2022
          imagePullPolicy: Always
          name: iis
      nodeSelector:
        kubernetes.io/os: windows
```

Individual containers in a Pod spec can also specify the desired GMSA credspec using a per-container `securityContext.windowsOptions.gmsaCredentialSpecName` field. For example:

```

apiVersion: apps/v1
kind: Deployment
metadata:
  labels:
    run: with-creds
  name: with-creds
  namespace: default
spec:
  replicas: 1
  selector:
    matchLabels:
      run: with-creds
  template:
    metadata:
      labels:
        run: with-creds
    spec:
      containers:
        - image: mcr.microsoft.com/windows/servercore/iis:windowsservercore-ltsc2019
          imagePullPolicy: Always
          name: iis
          securityContext:
            windowsOptions:
              gmsaCredentialSpecName: gmsa-Webapp1
      nodeSelector:
        kubernetes.io/os: windows

```

As Pod specs with GMSA fields populated (as described above) are applied in a cluster, the following sequence of events take place:

1. The mutating webhook resolves and expands all references to GMSA credential spec resources to the contents of the GMSA credential spec.
2. The validating webhook ensures the service account associated with the Pod is authorized for the `use` verb on the specified GMSA credential spec.
3. The container runtime configures each Windows container with the specified GMSA credential spec so that the container can assume the identity of the GMSA in Active Directory and access services in the domain using that identity.

Authenticating to network shares using hostname or FQDN

If you are experiencing issues connecting to SMB shares from Pods using hostname or FQDN, but are able to access the shares via their IPv4 address then make sure the following registry key is set on the Windows nodes.

```
reg add "HKLM\SYSTEM\CurrentControlSet\Services\hns\State" /v EnableCoreProtocol /t REG_DWORD /d 1
```

Running Pods will then need to be recreated to pick up the behavior changes. More information on how this registry key is used can be found [here](#)

Troubleshooting

If you are having difficulties getting GMSA to work in your environment, there are a few troubleshooting steps you can take.

First, make sure the credspec has been passed to the Pod. To do this you will need to `exec` into one of your Pods and check the output of the `nltest.exe /parentdomain` command.

In the example below the Pod did not get the credspec correctly:

```
kubectl exec -it iis-auth-7776966999-n5nzb powershell.exe
```

`nltest.exe /parentdomain` results in the following error:

```
Getting parent domain failed: Status = 1722 0x6ba RPC_S_SERVER_UNAVAILABLE
```

If your Pod did get the credspec correctly, then next check communication with the domain. First, from inside of your Pod, quickly do an nslookup to find the root of your domain.

This will tell us 3 things:

1. The Pod can reach the DC
2. The DC can reach the Pod
3. DNS is working correctly.

If the DNS and communication test passes, next you will need to check if the Pod has established secure channel communication with the domain. To do this, again, `exec` into your Pod and run the `nltest.exe /query` command.

```
nltest.exe /query
```

Results in the following output:

```
I_NetLogonControl failed: Status = 1722 0x6ba RPC_S_SERVER_UNAVAILABLE
```

This tells us that for some reason, the Pod was unable to logon to the domain using the account specified in the credspec. You can try to repair the secure channel by running the following:

```
nltest /sc_reset:domain.example
```

If the command is successful you will see and output similar to this:

```
Flags: 30 HAS_IP HAS_TIMESERV
Trusted DC Name \\dc10.domain.example
Trusted DC Connection Status Status = 0 0x0 NERR_Success
The command completed successfully
```

If the above corrects the error, you can automate the step by adding the following lifecycle hook to your Pod spec. If it did not correct the error, you will need to examine your credspec again and confirm that it is correct and

complete.

```
image: registry.domain.example/iis-auth:1809v1
lifecycle:
  postStart:
    exec:
      command: ["powershell.exe", "-command", "do { Restart-Serv
imagePullPolicy: IfNotPresent
```

If you add the `lifecycle` section show above to your Pod spec, the Pod will execute the commands listed to restart the `netlogon` service until the `nlttest.exe /query` command exits without error.

3.4 - Resize CPU and Memory Resources assigned to Containers

FEATURE STATE: [Kubernetes v1.27 \[alpha\]](#)

This page assumes that you are familiar with [Quality of Service](#) for Kubernetes Pods.

This page shows how to resize CPU and memory resources assigned to containers of a running pod without restarting the pod or its containers. A Kubernetes node allocates resources for a pod based on its `requests`, and restricts the pod's resource usage based on the `limits` specified in the pod's containers.

For in-place resize of pod resources:

- Container's resource `requests` and `limits` are *mutable* for CPU and memory resources.
- `allocatedResources` field in `containerStatuses` of the Pod's status reflects the resources allocated to the pod's containers.
- `resources` field in `containerStatuses` of the Pod's status reflects the actual resource `requests` and `limits` that are configured on the running containers as reported by the container runtime.
- `resize` field in the Pod's status shows the status of the last requested pending resize. It can have the following values:
 - `Proposed` : This value indicates an acknowledgement of the requested resize and that the request was validated and recorded.
 - `InProgress` : This value indicates that the node has accepted the resize request and is in the process of applying it to the pod's containers.
 - `Deferred` : This value means that the requested resize cannot be granted at this time, and the node will keep retrying. The resize may be granted when other pods leave and free up node resources.
 - `Infeasible` : is a signal that the node cannot accommodate the requested resize. This can happen if the requested resize exceeds the maximum resources the node can ever allocate for a pod.

Before you begin

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

- [Killercoda](#)
- [Play with Kubernetes](#)

Your Kubernetes server must be at or later than version 1.27. To check the version, enter [kubectl version](#).

Container Resize Policies

Resize policies allow for a more fine-grained control over how pod's containers are resized for CPU and memory resources. For example, the container's application may be able to handle CPU resources resized without being restarted, but resizing memory may require that the application hence the containers be restarted.

To enable this, the Container specification allows users to specify a `resizePolicy`. The following restart policies can be specified for resizing CPU and memory:

- `NotRequired` : Resize the container's resources while it is running.
- `RestartContainer` : Restart the container and apply new resources upon restart.

If `resizePolicy[*].restartPolicy` is not specified, it defaults to `NotRequired`.

Note: If the Pod's `restartPolicy` is `Never`, container's resize restart policy must be set to `NotRequired` for all Containers in the Pod.

Below example shows a Pod whose Container's CPU can be resized without restart, but resizing memory requires the container to be restarted.

```
apiVersion: v1
kind: Pod
metadata:
  name: qos-demo-5
  namespace: qos-example
spec:
  containers:
    - name: qos-demo-ctr-5
      image: nginx
      resizePolicy:
        - resourceName: cpu
          restartPolicy: NotRequired
        - resourceName: memory
          restartPolicy: RestartContainer
      resources:
        limits:
          memory: "200Mi"
          cpu: "700m"
        requests:
          memory: "200Mi"
          cpu: "700m"
```

Note: In the above example, if desired requests or limits for both CPU and memory have changed, the container will be restarted in order to resize its memory.

Create a pod with resource requests and limits

You can create a Guaranteed or Burstable [Quality of Service](#) class pod by specifying requests and/or limits for a pod's containers.

Consider the following manifest for a Pod that has one Container.

[pods/qos/qos-pod-5.yaml](#) 

```
apiVersion: v1
kind: Pod
metadata:
  name: qos-demo-5
  namespace: qos-example
spec:
  containers:
  - name: qos-demo-ctr-5
    image: nginx
    resources:
      limits:
        memory: "200Mi"
        cpu: "700m"
      requests:
        memory: "200Mi"
        cpu: "700m"
```

Create the pod in the `qos-example` namespace:

```
kubectl create namespace qos-example
kubectl create -f https://k8s.io/examples/pods/qos/qos-pod-5.yaml
```

This pod is classified as a Guaranteed QoS class requesting 700m CPU and 200Mi memory.

View detailed information about the pod:

```
kubectl get pod qos-demo-5 --output=yaml --namespace=qos-example
```

Also notice that the values of `resizePolicy[*].restartPolicy` defaulted to `NotRequired`, indicating that CPU and memory can be resized while container is running.

```
spec:  
  containers:  
    ...  
    resizePolicy:  
      - resourceName: cpu  
        restartPolicy: NotRequired  
      - resourceName: memory  
        restartPolicy: NotRequired  
    resources:  
      limits:  
        cpu: 700m  
        memory: 200Mi  
      requests:  
        cpu: 700m  
        memory: 200Mi  
    ...  
  containerStatuses:  
    ...  
    name: qos-demo-ctr-5  
    ready: true  
    ...  
  allocatedResources:  
    cpu: 700m  
    memory: 200Mi  
  resources:  
    limits:  
      cpu: 700m  
      memory: 200Mi  
    requests:  
      cpu: 700m  
      memory: 200Mi  
  restartCount: 0  
  started: true  
...  
qosClass: Guaranteed
```

Updating the pod's resources

Let's say the CPU requirements have increased, and 0.8 CPU is now desired. This is typically determined, and may be programmatically applied, by an entity such as [VerticalPodAutoscaler](#) (VPA).

Note: While you can change a Pod's requests and limits to express new desired resources, you cannot change the QoS class in which the Pod was created.

Now, patch the Pod's Container with CPU requests & limits both set to 800m :

```
kubectl -n qos-example patch pod qos-demo-5 --patch '{"spec":{"containers": [{"resources": {"limits": {"cpu": "800m"}, "requests": {"cpu": "800m"}}, "restartPolicy": "Never"}]}}'
```

Query the Pod's detailed information after the Pod has been patched.

```
kubectl get pod qos-demo-5 --output=yaml --namespace=qos-example
```

The Pod's spec below reflects the updated CPU requests and limits.

```
spec:  
  containers:  
    ...  
    resources:  
      limits:  
        cpu: 800m  
        memory: 200Mi  
      requests:  
        cpu: 800m  
        memory: 200Mi  
    ...  
    containerStatuses:  
    ...  
      allocatedResources:  
        cpu: 800m  
        memory: 200Mi  
      resources:  
        limits:  
          cpu: 800m  
          memory: 200Mi  
        requests:  
          cpu: 800m  
          memory: 200Mi  
      restartCount: 0  
      started: true
```

Observe that the `allocatedResources` values have been updated to reflect the new desired CPU requests. This indicates that node was able to accommodate the increased CPU resource needs.

In the Container's status, updated CPU resource values shows that new CPU resources have been applied. The Container's `restartCount` remains unchanged, indicating that container's CPU resources were resized without restarting the container.

Clean up

Delete your namespace:

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

What's next

For application 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](#)

3.5 - Configure RunAsUserName for Windows pods and containers

FEATURE STATE: [Kubernetes v1.18 \[stable\]](#)

This page shows how to use the `runAsUserName` setting for Pods and containers that will run on Windows nodes. This is roughly equivalent of the Linux-specific `runAsUser` setting, allowing you to run applications in a container as a different username than the default.

Before you begin

You need to have a Kubernetes cluster and the `kubectl` command-line tool must be configured to communicate with your cluster. The cluster is expected to have Windows worker nodes where pods with containers running Windows workloads will get scheduled.

Set the Username for a Pod

To specify the username with which to execute the Pod's container processes, include the `securityContext` field ([PodSecurityContext](#)) in the Pod specification, and within it, the `windowsOptions` ([WindowsSecurityContextOptions](#)) field containing the `runAsUserName` field.

The Windows security context options that you specify for a Pod apply to all Containers and init Containers in the Pod.

Here is a configuration file for a Windows Pod that has the `runAsUserName` field set:

```
windows/run-as-username-pod.yaml 
```

```
apiVersion: v1
kind: Pod
metadata:
  name: run-as-username-pod-demo
spec:
  securityContext:
    windowsOptions:
      runAsUserName: "ContainerUser"
  containers:
  - name: run-as-username-demo
    image: mcr.microsoft.com/windows/servercore:ltsc2019
    command: ["ping", "-t", "localhost"]
  nodeSelector:
    kubernetes.io/os: windows
```

Create the Pod:

```
kubectl apply -f https://k8s.io/examples/windows/run-as-username-pod.yaml
```

Verify that the Pod's Container is running:

```
kubectl get pod run-as-username-pod-demo
```

Get a shell to the running Container:

```
kubectl exec -it run-as-username-pod-demo -- powershell
```

Check that the shell is running user the correct username:

```
echo $env:USERNAME
```

The output should be:

```
ContainerUser
```

Set the Username for a Container

To specify the username with which to execute a Container's processes, include the `securityContext` field ([SecurityContext](#)) in the Container manifest, and within it, the `windowsOptions` ([WindowsSecurityContextOptions](#)) field containing the `runAsUserName` field.

The Windows security context options that you specify for a Container apply only to that individual Container, and they override the settings made at the Pod level.

Here is the configuration file for a Pod that has one Container, and the `runAsUserName` field is set at the Pod level and the Container level:

[windows/run-as-username-container.yaml](#) 

```
apiVersion: v1
kind: Pod
metadata:
  name: run-as-username-container-demo
spec:
  securityContext:
    windowsOptions:
      runAsUserName: "ContainerUser"
  containers:
  - name: run-as-username-demo
    image: mcr.microsoft.com/windows/servercore:ltsc2019
    command: ["ping", "-t", "localhost"]
    securityContext:
      windowsOptions:
        runAsUserName: "ContainerAdministrator"
  nodeSelector:
    kubernetes.io/os: windows
```

Create the Pod:

```
kubectl apply -f https://k8s.io/examples/windows/run-as-username-conta:
```

Verify that the Pod's Container is running:

```
kubectl get pod run-as-username-container-demo
```

Get a shell to the running Container:

```
kubectl exec -it run-as-username-container-demo -- powershell
```

Check that the shell is running user the correct username (the one set at the Container level):

```
echo $env:USERNAME
```

The output should be:

```
ContainerAdministrator
```

Windows Username limitations

In order to use this feature, the value set in the `runAsUserName` field must be a valid username. It must have the following format: `DOMAIN\USER`, where `DOMAIN\` is optional. Windows user names are case insensitive.

Additionally, there are some restrictions regarding the `DOMAIN` and `USER`:

- The `runAsUserName` field cannot be empty, and it cannot contain control characters (ASCII values: `0x00-0x1F`, `0x7F`)
- The `DOMAIN` must be either a NetBios name, or a DNS name, each with their own restrictions:
 - NetBios names: maximum 15 characters, cannot start with `.` (dot), and cannot contain the following characters: `\ / : * ? " < > |`
 - DNS names: maximum 255 characters, contains only alphanumeric characters, dots, and dashes, and it cannot start or end with a `.` (dot) or `-` (dash).
- The `USER` must have at most 20 characters, it cannot contain *only* dots or spaces, and it cannot contain the following characters: `" / \ [] : ; | = , + * ? < > @ .`

Examples of acceptable values for the `runAsUserName` field:

`ContainerAdministrator` , `ContainerUser` , `NT AUTHORITY\NETWORK SERVICE` , `NT AUTHORITY\LOCAL SERVICE` .

For more information about these limitations, check [here](#) and [here](#).

What's next

- [Guide for scheduling Windows containers in Kubernetes](#)
- [Managing Workload Identity with Group Managed Service Accounts \(GMSA\)](#)

- [Configure GMSA for Windows pods and containers](#)

3.6 - Create a Windows HostProcess Pod

FEATURE STATE: [Kubernetes v1.26 \[stable\]](#)

Windows HostProcess containers enable you to run containerized workloads on a Windows host. These containers operate as normal processes but have access to the host network namespace, storage, and devices when given the appropriate user privileges. HostProcess containers can be used to deploy network plugins, storage configurations, device plugins, kube-proxy, and other components to Windows nodes without the need for dedicated proxies or the direct installation of host services.

Administrative tasks such as installation of security patches, event log collection, and more can be performed without requiring cluster operators to log onto each Windows node. HostProcess containers can run as any user that is available on the host or is in the domain of the host machine, allowing administrators to restrict resource access through user permissions. While neither filesystem or process isolation are supported, a new volume is created on the host upon starting the container to give it a clean and consolidated workspace. HostProcess containers can also be built on top of existing Windows base images and do not inherit the same [compatibility requirements](#) as Windows server containers, meaning that the version of the base images does not need to match that of the host. It is, however, recommended that you use the same base image version as your Windows Server container workloads to ensure you do not have any unused images taking up space on the node. HostProcess containers also support [volume mounts](#) within the container volume.

When should I use a Windows HostProcess container?

- When you need to perform tasks which require the networking namespace of the host. HostProcess containers have access to the host's network interfaces and IP addresses.
- You need access to resources on the host such as the filesystem, event logs, etc.
- Installation of specific device drivers or Windows services.
- Consolidation of administrative tasks and security policies. This reduces the degree of privileges needed by Windows nodes.

Before you begin

This task guide is specific to Kubernetes v1.28. If you are not running Kubernetes v1.28, check the documentation for that version of Kubernetes.

In Kubernetes 1.28, the HostProcess container feature is enabled by default. The kubelet will communicate with containerd directly by passing the hostprocess flag via CRI. You can use the latest version of containerd (v1.6+) to run HostProcess containers. [How to install containerd](#).

Limitations

These limitations are relevant for Kubernetes v1.28:

- HostProcess containers require containerd 1.6 or higher container runtime and containerd 1.7 is recommended.

- HostProcess pods can only contain HostProcess containers. This is a current limitation of the Windows OS; non-privileged Windows containers cannot share a vNIC with the host IP namespace.
- HostProcess containers run as a process on the host and do not have any degree of isolation other than resource constraints imposed on the HostProcess user account. Neither filesystem or Hyper-V isolation are supported for HostProcess containers.
- Volume mounts are supported and are mounted under the container volume. See [Volume Mounts](#)
- A limited set of host user accounts are available for HostProcess containers by default. See [Choosing a User Account](#).
- Resource limits (disk, memory, cpu count) are supported in the same fashion as processes on the host.
- Both Named pipe mounts and Unix domain sockets are **not** supported and should instead be accessed via their path on the host (e.g. \\.\pipe*)

HostProcess Pod configuration requirements

Enabling a Windows HostProcess pod requires setting the right configurations in the pod security configuration. Of the policies defined in the [Pod Security Standards](#) HostProcess pods are disallowed by the baseline and restricted policies. It is therefore recommended that HostProcess pods run in alignment with the privileged profile.

When running under the privileged policy, here are the configurations which need to be set to enable the creation of a HostProcess pod:

Control	Policy
securityContext.windowsOptions.hostProcess	<p>Windows pods offer the ability to run HostProcess containers which enables privileged access to the Windows node.</p> <p>Allowed Values</p> <ul style="list-style-type: none">• true
hostNetwork	<p>Pods containing HostProcess containers must use the host's network namespace.</p> <p>Allowed Values</p> <ul style="list-style-type: none">• true

Control	Policy
<code>securityContext.windowsOptions.runAsUserName</code>	Specification of which user the HostProcess container should run as is required for the pod spec.
<code>runAsNonRoot</code>	Because HostProcess containers have privileged access to the host, the <code>runAsNonRoot</code> field cannot be set to true.

Allowed Values

- NT AUTHORITY\SYSTEM
- NT AUTHORITY\Local service
- NT AUTHORITY\NetworkService
- Local usergroup names (see below)

Because HostProcess containers have privileged access to the host, the `runAsNonRoot` field cannot be set to true.

Allowed Values

- Undefined/Nil
- false

Example manifest (excerpt)

```
spec:
  securityContext:
    windowsOptions:
      hostProcess: true
      runAsUserName: "NT AUTHORITY\Local service"
    hostNetwork: true
  containers:
  - name: test
    image: image1:latest
    command:
    - ping
    - -t
    - 127.0.0.1
  nodeSelector:
    "kubernetes.io/os": windows
```

Volume mounts

HostProcess containers support the ability to mount volumes within the container volume space. Volume mount behavior differs depending on the version of containerd runtime used by on the node.

Containerd v1.6

Applications running inside the container can access volume mounts directly via relative or absolute paths. An environment variable `$CONTAINER_SANDBOX_MOUNT_POINT` is set upon container creation and provides the absolute host path to the container volume. Relative paths are based upon the `.spec.containers.volumeMounts.mountPath` configuration.

To access service account tokens (for example) the following path structures are supported within the container:

- .\var\run\secrets\kubernetes.io\serviceaccount\
- \$CONTAINER_SANDBOX_MOUNT_POINT\var\run\secrets\kubernetes.io\serviceaccount\

Containerd v1.7 (and greater)

Applications running inside the container can access volume mounts directly via the volumeMount's specified `mountPath` (just like Linux and non-HostProcess Windows containers).

For backwards compatibility volumes can also be accessed via using the same relative paths configured by containerd v1.6.

As an example, to access service account tokens within the container you would use one of the following paths:

- c:\var\run\secrets\kubernetes.io\serviceaccount\
- /var/run/secrets/kubernetes.io/serviceaccount/
- \$CONTAINER_SANDBOX_MOUNT_POINT\var\run\secrets\kubernetes.io\serviceaccount\

Resource limits

Resource limits (disk, memory, cpu count) are applied to the job and are job wide. For example, with a limit of 10MB set, the memory allocated for any HostProcess job object will be capped at 10MB. This is the same behavior as other Windows container types. These limits would be specified the same way they are currently for whatever orchestrator or runtime is being used. The only difference is in the disk resource usage calculation used for resource tracking due to the difference in how HostProcess containers are bootstrapped.

Choosing a user account

System accounts

By default, HostProcess containers support the ability to run as one of three supported Windows service accounts:

- [LocalSystem](#)
- [LocalService](#)
- [NetworkService](#)

You should select an appropriate Windows service account for each HostProcess container, aiming to limit the degree of privileges so as to avoid accidental (or even malicious) damage to the host. The LocalSystem service account has the highest level of privilege of the three and should be used only if absolutely necessary. Where possible, use the LocalService service account as it is the least privileged of the three options.

Local accounts

If configured, HostProcess containers can also run as local user accounts which allows for node operators to give fine-grained access to workloads.

To run HostProcess containers as a local user; A local usergroup must first be created on the node and the name of that local usergroup must be specified in the `runAsUserName` field in the deployment. Prior to initializing

the HostProcess container, a new **ephemeral** local user account to be created and joined to the specified usergroup, from which the container is run. This provides a number of benefits including eliminating the need to manage passwords for local user accounts. An initial HostProcess container running as a service account can be used to prepare the user groups for later HostProcess containers.

Note: Running HostProcess containers as local user accounts requires containerd v1.7+

Example:

1. Create a local user group on the node (this can be done in another HostProcess container).

```
net localgroup hpc-localgroup /add
```

2. Grant access to desired resources on the node to the local usergroup. This can be done with tools like [icacls](#).

3. Set `runAsUserName` to the name of the local usergroup for the pod or individual containers.

```
securityContext:  
  windowsOptions:  
    hostProcess: true  
    runAsUserName: hpc-localgroup
```

4. Schedule the pod!

Base Image for HostProcess Containers

HostProcess containers can be built from any of the existing [Windows Container base images](#).

Additionally a new base image has been created just for HostProcess containers! For more information please check out the [windows-host-process-containers-base-image github project](#).

Troubleshooting HostProcess containers

- HostProcess containers fail to start with `failed to create user process token: failed to logon user: Access is denied.: unknown`

Ensure containerd is running as `LocalSystem` or `LocalService` service accounts. User accounts (even Administrator accounts) do not have permissions to create logon tokens for any of the supported [user accounts](#).

3.7 - 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 evicting Pods when Node resources are exceeded.

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

- [Guaranteed](#)
- [Burstable](#)
- [BestEffort](#)

Before you begin

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

- [KillerCoda](#)
- [Play with Kubernetes](#)

You also need to be able to create and delete namespaces.

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.
- For every Container in the Pod, the memory limit must equal the memory request.
- Every Container in the Pod must have a CPU limit and a CPU request.
- For every Container in the Pod, the CPU limit must equal the CPU request.

These restrictions apply to init containers and app containers equally.

[Ephemeral containers](#) cannot define resources so these restrictions do not apply.

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

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

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

Create the Pod:

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

View detailed information about the Pod:

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

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

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

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

Clean up

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 or limit.

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

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

Create the Pod:

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

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  
...  
status:  
  qosClass: Burstable
```

Clean up

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 a manifest for a Pod that has one Container. The Container has no memory or CPU limits or requests:

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

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

Create the Pod:

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

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: {}  
    ...  
  status:  
    qosClass: BestEffort
```

Clean up

Delete your Pod:

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

Create a Pod that has two Containers

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

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

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

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

Create the Pod:

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

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: {}  
  ...  
status:  
  qosClass: Burstable
```

Retrieve the QoS class for a Pod

Rather than see all the fields, you can view just the field you need:

```
kubectl --namespace=qos-example get pod qos-demo-4 -o jsonpath='{ .sta
```

```
◀ ▶
```

```
Burstable
```

Clean up

Delete your namespace:

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

What's next

For app developers

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

For cluster administrators

- [Configure Default Memory Requests and Limits for a Namespace](#)
- [Configure Default CPU Requests and Limits for a Namespace](#)
- [Configure Minimum and Maximum Memory Constraints for a Namespace](#)
- [Configure Minimum and Maximum CPU Constraints for a Namespace](#)
- [Configure Memory and CPU Quotas for a Namespace](#)
- [Configure a Pod Quota for a Namespace](#)
- [Configure Quotas for API Objects](#)

- [Control Topology Management policies on a node](#)

3.8 - Assign Extended Resources to a Container

FEATURE STATE: [Kubernetes v1.28 \[stable\]](#)

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

Before you begin

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

- [Killercoda](#)
- [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:

```
pods/resource/extended-resource-pod.yaml 
```

```
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 apply -f https://k8s.io/examples/pods/resource/extended-resource
```

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.

```
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 apply -f https://k8s.io/examples/pods/resource/extended-resource
```

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 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 Pods that you created for this exercise:

```
kubectl delete pod extended-resource-demo  
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](#)

3.9 - Configure a Pod to Use a Volume for Storage

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

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

Before you begin

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

- [Killercoda](#)
- [Play with Kubernetes](#)

To check the version, enter `kubectl version`.

Configure a volume for a Pod

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

```
pods/storage/redis.yaml 
```

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

1. Create the Pod:

```
kubectl apply -f https://k8s.io/examples/pods/storage/redis.yaml
```

2. Verify that the Pod's Container is running, and then watch for changes to the Pod:

```
kubectl get pod redis --watch
```

The output looks like this:

NAME	READY	STATUS	RESTARTS	AGE
redis	1/1	Running	0	13s

3. In another terminal, get a shell to the running Container:

```
kubectl exec -it redis -- /bin/bash
```

4. In your shell, go to `/data/redis`, and then create a file:

```
root@redis:/data# cd /data/redis/  
root@redis:/data/redis# echo Hello > test-file
```

5. In your shell, list the running processes:

```
root@redis:/data/redis# apt-get update  
root@redis:/data/redis# apt-get install procps  
root@redis:/data/redis# ps aux
```

The output is similar to this:

USER	PID	%CPU	%MEM	VSZ	RSS	TTY	STAT	START	TIME
redis	1	0.1	0.1	33308	3828	?	Ssl	00:46	0:00
root	12	0.0	0.0	20228	3020	?	Ss	00:47	0:00
root	15	0.0	0.0	17500	2072	?	R+	00:48	0:00

6. In your shell, kill the Redis process:

```
root@redis:/data/redis# kill <pid>
```

where `<pid>` is the Redis process ID (PID).

7. In your original terminal, watch for changes to the Redis Pod. Eventually, you will see something like this:

NAME	READY	STATUS	RESTARTS	AGE
redis	1/1	Running	0	13s
redis	0/1	Completed	0	6m
redis	1/1	Running	1	6m

At this point, the Container has terminated and restarted. This is because the Redis Pod has a [restartPolicy](#) of `Always`.

1. Get a shell into the restarted Container:

```
kubectl exec -it redis -- /bin/bash
```

2. In your shell, go to `/data/redis`, and verify that `test-file` is still there.

```
root@redis:/data/redis# cd /data/redis/
root@redis:/data/redis# ls
test-file
```

3. Delete the Pod that you created for this exercise:

```
kubectl delete pod redis
```

What's next

- See [Volume](#).
- See [Pod](#).
- In addition to the local disk storage provided by `emptyDir`, Kubernetes supports many different network-attached storage solutions, including PD on GCE and EBS on EC2, which are preferred for critical data and will handle details such as mounting and unmounting the devices on the nodes. See [Volumes](#) for more details.

3.10 - Configure a Pod to Use a PersistentVolume for Storage

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

1. You, as cluster administrator, create a `PersistentVolume` backed by physical storage. You do not associate the volume with any Pod.
2. You, now taking the role of a developer / cluster user, create a `PersistentVolumeClaim` that is automatically bound to a suitable `PersistentVolume`.
3. You create a Pod that uses the above `PersistentVolumeClaim` for storage.

Before you begin

- 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 single Node in your cluster. How you open a shell depends on how you set up your cluster. For example, if you are using Minikube, you can open a shell to your Node by entering `minikube ssh`.

In your shell on that Node, create a `/mnt/data` directory:

```
# This assumes that your Node uses "sudo" to run commands
# as the superuser
sudo mkdir /mnt/data
```

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

```
# This again assumes that your Node uses "sudo" to run commands
# as the superuser
sudo sh -c "echo 'Hello from Kubernetes storage' > /mnt/data/index.html"
```

Note: If your Node uses a tool for superuser access other than `sudo`, you can usually make this work if you replace `sudo` with the name of the other tool.

Test that the `index.html` file exists:

```
cat /mnt/data/index.html
```

The output should be:

```
Hello from Kubernetes storage
```

You can now close the shell to your Node.

Create a PersistentVolume

In this exercise, you create a *hostPath* PersistentVolume. Kubernetes supports hostPath for development and testing on a single-node cluster. A hostPath PersistentVolume uses a file or directory on the Node to emulate network-attached storage.

In a production cluster, you would not use hostPath. Instead a cluster administrator would provision a network resource like a Google Compute Engine persistent disk, an NFS share, or an Amazon Elastic Block Store volume. Cluster administrators can also use [StorageClasses](#) to set up [dynamic provisioning](#).

Here is the configuration file for the hostPath PersistentVolume:

[pods/storage/pv-volume.yaml](#) 

```
apiVersion: v1
kind: PersistentVolume
metadata:
  name: task-pv-volume
  labels:
    type: local
spec:
  storageClassName: manual
  capacity:
    storage: 10Gi
  accessModes:
    - ReadWriteOnce
  hostPath:
    path: "/mnt/data"
```

The configuration file specifies that the volume is at `/mnt/data` on the cluster's Node. The configuration also specifies a size of 10 gibibytes and an access mode of `ReadWriteOnce`, which means the volume can be mounted as read-write by a single Node. It defines the [StorageClass name](#) `manual` for the PersistentVolume, which will be used to bind PersistentVolumeClaim requests to this PersistentVolume.

Create the PersistentVolume:

```
kubectl apply -f https://k8s.io/examples/pods/storage/pv-volume.yaml
```

View information about the PersistentVolume:

```
kubectl get pv task-pv-volume
```

The output shows that the PersistentVolume has a STATUS of Available . This means it has not yet been bound to a PersistentVolumeClaim.

NAME	CAPACITY	ACCESSMODES	RECLAIMPOLICY	STATUS	CLASS
task-pv-volume	10Gi	RWO	Retain	Available	default

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 most one Node at a time.

Here is the configuration file for the PersistentVolumeClaim:

[pods/storage/pv-claim.yaml](#) 

```

apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: task-pv-claim
spec:
  storageClassName: manual
  accessModes:
    - ReadWriteOnce
  resources:
    requests:
      storage: 3Gi

```

Create the PersistentVolumeClaim:

```
kubectl apply -f https://k8s.io/examples/pods/storage/pv-claim.yaml
```

After you create the PersistentVolumeClaim, the Kubernetes control plane looks for a PersistentVolume that satisfies the claim's requirements. If the control plane finds a suitable PersistentVolume with the same StorageClass, it binds the claim to the volume.

Look again at the PersistentVolume:

```
kubectl get pv task-pv-volume
```

Now the output shows a STATUS of Bound .

NAME	CAPACITY	ACCESSMODES	RECLAIMPOLICY	STATUS	CLASS
task-pv-volume	10Gi	RWO	Retain	Bound	default

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	STC
task-pv-claim	Bound	task-pv-volume	10Gi	RWO	mar

◀ ▶

Create a Pod

The next step is to create a Pod that uses your PersistentVolumeClaim as a volume.

Here is the configuration file for the Pod:

[pods/storage/pv-pod.yaml](#)

```
apiVersion: v1
kind: Pod
metadata:
  name: task-pv-pod
spec:
  volumes:
    - name: task-pv-storage
      persistentVolumeClaim:
        claimName: task-pv-claim
  containers:
    - name: task-pv-container
      image: nginx
      ports:
        - containerPort: 80
          name: "http-server"
      volumeMounts:
        - mountPath: "/usr/share/nginx/html"
          name: task-pv-storage
```

Notice that the Pod's configuration file specifies a PersistentVolumeClaim, but it does not specify a PersistentVolume. From the Pod's point of view, the claim is a volume.

Create the Pod:

```
kubectl apply -f https://k8s.io/examples/pods/storage/pv-pod.yaml
```

Verify that the container in the Pod is running:

```
kubectl get pod task-pv-pod
```

Get a shell to the container running in your Pod:

```
kubectl exec -it task-pv-pod -- /bin/bash
```

In your shell, verify that nginx is serving the `index.html` file from the hostPath volume:

```
# Be sure to run these 3 commands inside the root shell that comes from
# running "kubectl exec" in the previous step
apt update
apt install curl
curl http://localhost/
```

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

```
Hello from Kubernetes storage
```

If you see that message, you have successfully configured a Pod to use storage from a PersistentVolumeClaim.

Clean up

Delete the Pod, the PersistentVolumeClaim and the PersistentVolume:

```
kubectl delete pod task-pv-pod
kubectl delete pvc task-pv-claim
kubectl delete pv task-pv-volume
```

If you don't already have a shell open to the Node in your cluster, open a new shell the same way that you did earlier.

In the shell on your Node, remove the file and directory that you created:

```
# This assumes that your Node uses "sudo" to run commands
# as the superuser
sudo rm /mnt/data/index.html
sudo rmdir /mnt/data
```

You can now close the shell to your Node.

Mounting the same persistentVolume in two places

[pods/storage/pv-duplicate.yaml](#) 

```
apiVersion: v1
kind: Pod
metadata:
  name: test
spec:
  containers:
    - name: test
      image: nginx
      volumeMounts:
        # a mount for site-data
        - name: config
          mountPath: /usr/share/nginx/html
          subPath: html
        # another mount for nginx config
        - name: config
          mountPath: /etc/nginx/nginx.conf
          subPath: nginx.conf
  volumes:
    - name: config
      persistentVolumeClaim:
        claimName: test-nfs-claim
```

You can perform 2 volume mounts on your nginx container:

- `/usr/share/nginx/html` for the static website
- `/etc/nginx/nginx.conf` for the default config

Access control

Storage configured with a group ID (GID) allows writing only by Pods using the same GID. Mismatched or missing GIDs cause permission denied errors. To reduce the need for coordination with users, an administrator can annotate a PersistentVolume with a GID. Then the GID is automatically added to any Pod that uses the PersistentVolume.

Use the `pv.beta.kubernetes.io/gid` annotation as follows:

```
apiVersion: v1
kind: PersistentVolume
metadata:
  name: pv1
  annotations:
    pv.beta.kubernetes.io/gid: "1234"
```

When a Pod consumes a PersistentVolume that has a GID annotation, the annotated GID is applied to all containers in the Pod in the same way that GIDs specified in the Pod's security context are. Every GID, whether it originates from a PersistentVolume annotation or the Pod's specification, is applied to the first process run in each container.

Note: When a Pod consumes a PersistentVolume, the GIDs associated with the PersistentVolume are not present on the Pod resource itself.

What's next

- Learn more about [PersistentVolumes](#).
- Read the [Persistent Storage design document](#).

Reference

- [PersistentVolume](#)
- [PersistentVolumeSpec](#)
- [PersistentVolumeClaim](#)
- [PersistentVolumeClaimSpec](#)

3.11 - Configure a Pod to Use a Projected Volume for Storage

This page shows how to use a [projected](#) Volume to mount several existing volume sources into the same directory. Currently, `secret`, `configMap`, `downwardAPI`, and `serviceAccountToken` volumes can be projected.

Note: `serviceAccountToken` is not a volume type.

Before you begin

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

- [Killercoda](#)
- [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:

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

```
apiVersion: v1
kind: Pod
metadata:
  name: test-projected-volume
spec:
  containers:
    - name: test-projected-volume
      image: busybox:1.28
      args:
        - sleep
        - "86400"
      volumeMounts:
        - name: all-in-one
          mountPath: "/projected-volume"
          readOnly: true
  volumes:
    - name: all-in-one
      projected:
        sources:
          - secret:
              name: user
          - secret:
              name: pass
```

1. Create the Secrets:

```
# Create files containing the username and password:
echo -n "admin" > ./username.txt
echo -n "1f2d1e2e67df" > ./password.txt

# Package these files into secrets:
kubectl create secret generic user --from-file=./username.txt
kubectl create secret generic pass --from-file=./password.txt
```

2. Create the Pod:

```
kubectl apply -f https://k8s.io/examples/pods/storage/projected.y
```

3. Verify that the Pod's container is running, and then watch for changes to the Pod:

```
kubectl get --watch pod test-projected-volume
```

The output looks like this:

NAME	READY	STATUS	RESTARTS	AGE
test-projected-volume	1/1	Running	0	14s

4. In another terminal, get a shell to the running container:

```
kubectl exec -it test-projected-volume -- /bin/sh
```

5. In your shell, verify that the `projected-volume` directory contains your projected sources:

```
ls /projected-volume/
```

Clean up

Delete the Pod and the Secrets:

```
kubectl delete pod test-projected-volume  
kubectl delete secret user pass
```

What's next

- Learn more about [projected](#) volumes.
- Read the [all-in-one volume](#) design document.

3.12 - 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, but are not limited to:

- 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 always true when the container:
 - is run as privileged, or
 - has `CAP_SYS_ADMIN`
- `readOnlyRootFilesystem` : Mounts the container's root filesystem as read-only.

The above bullets are not a complete set of security context settings -- please see [SecurityContext](#) for a comprehensive list.

Before you begin

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

- [Killercoda](#)
- [Play with Kubernetes](#)

To check the version, enter `kubectl version`.

Set the security context for a Pod

To specify security settings for a Pod, include the `securityContext` field in the Pod specification. The `securityContext` field is a [PodSecurityContext](#) object. The security settings that you specify for a Pod apply to all Containers in the Pod. Here is a configuration file for a Pod that has a `securityContext` and an `emptyDir` volume:

[pods/security/security-context.yaml](#) 

```
apiVersion: v1
kind: Pod
metadata:
  name: security-context-demo
spec:
  securityContext:
    runAsUser: 1000
    runAsGroup: 3000
    fsGroup: 2000
  volumes:
  - name: sec-ctx-vol
    emptyDir: {}
  containers:
  - name: sec-ctx-demo
    image: busybox:1.28
    command: [ "sh", "-c", "sleep 1h" ]
    volumeMounts:
    - name: sec-ctx-vol
      mountPath: /data/demo
    securityContext:
      allowPrivilegeEscalation: false
```

In the configuration file, the `runAsUser` field specifies that for any Containers in the Pod, all processes run with user ID 1000. The `runAsGroup` field specifies the primary group ID of 3000 for all processes within any containers of the Pod. If this field is omitted, the primary group ID of the containers will be root(0). Any files created will also be owned by user 1000 and group 3000 when `runAsGroup` is specified. Since `fsGroup` field is specified, all processes of the container are also part of the supplementary group ID 2000. The owner for volume `/data/demo` and any files created in that volume will be Group ID 2000.

Create the Pod:

```
kubectl apply -f https://k8s.io/examples/pods/security/security-contex
```

Verify that the Pod's Container is running:

```
kubectl get pod security-context-demo
```

Get a shell to the running Container:

```
kubectl exec -it security-context-demo -- sh
```

In your shell, list the running processes:

```
ps
```

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

PID	USER	TIME	COMMAND
1	1000	0:00	sleep 1h
6	1000	0:00	sh
...			

In your shell, navigate to `/data`, and list the one directory:

```
cd /data  
ls -l
```

The output shows that the `/data/demo` directory has group ID 2000, which is the value of `fsGroup`.

```
drwxrwsrwx 2 root 2000 4096 Jun 6 20:08 demo
```

In your shell, navigate to `/data/demo`, and create a file:

```
cd demo  
echo hello > testfile
```

List the file in the `/data/demo` directory:

```
ls -l
```

The output shows that `testfile` has group ID 2000, which is the value of `fsGroup`.

```
-rw-r--r-- 1 1000 2000 6 Jun 6 20:08 testfile
```

Run the following command:

```
id
```

The output is similar to this:

```
uid=1000 gid=3000 groups=2000
```

From the output, you can see that `gid` is 3000 which is same as the `runAsGroup` field. If the `runAsGroup` was omitted, the `gid` would remain as 0 (root) and the process will be able to interact with files that are owned by the root(0) group and groups that have the required group permissions for the root (0) group.

Exit your shell:

```
exit
```

Configure volume permission and ownership change policy for Pods

FEATURE STATE: [Kubernetes v1.23 \[stable\]](#)

By default, Kubernetes recursively changes ownership and permissions for the contents of each volume to match the `fsGroup` specified in a Pod's `securityContext` when that volume is mounted. For large volumes, checking and changing ownership and permissions can take a lot of time, slowing Pod startup. You can use the `fsGroupChangePolicy` field inside a `securityContext` to control the way that Kubernetes checks and manages ownership and permissions for a volume.

fsGroupChangePolicy - `fsGroupChangePolicy` defines behavior for changing ownership and permission of the volume before being exposed inside a Pod. This field only applies to volume types that support `fsGroup` controlled ownership and permissions. This field has two possible values:

- *OnRootMismatch*: Only change permissions and ownership if the permission and the ownership of root directory does not match with expected permissions of the volume. This could help shorten the time it takes to change ownership and permission of a volume.
- *Always*: Always change permission and ownership of the volume when volume is mounted.

For example:

```
securityContext:  
  runAsUser: 1000  
  runAsGroup: 3000  
  fsGroup: 2000  
  fsGroupChangePolicy: "OnRootMismatch"
```

Note: This field has no effect on ephemeral volume types such as [secret](#), [configMap](#), and [emptydir](#).

Delegating volume permission and ownership change to CSI driver

FEATURE STATE: [Kubernetes v1.26 \[stable\]](#)

If you deploy a [Container Storage Interface \(CSI\)](#) driver which supports the `VOLUME_MOUNT_GROUP` `NodeServiceCapability`, the process of setting file ownership and permissions based on the `fsGroup` specified in the `securityContext` will be performed by the CSI driver instead of Kubernetes. In this case, since Kubernetes doesn't perform any ownership and permission change, `fsGroupChangePolicy` does not take effect, and as specified by CSI, the driver is expected to mount the volume with the provided `fsGroup`, resulting in a volume that is readable/writable by the `fsGroup`.

Set the security context for a Container

To specify security settings for a Container, include the `securityContext` field in the Container manifest. The `securityContext` field is a [SecurityContext](#) object. Security settings that you specify for a Container

apply only to the individual Container, and they override settings made at the Pod level when there is overlap. Container settings do not affect the Pod's Volumes.

Here is the configuration file for a Pod that has one Container. Both the Pod and the Container have a `securityContext` field:

[pods/security/security-context-2.yaml](#) 

```
apiVersion: v1
kind: Pod
metadata:
  name: security-context-demo-2
spec:
  securityContext:
    runAsUser: 1000
  containers:
  - name: sec-ctx-demo-2
    image: gcr.io/google-samples/node-hello:1.0
    securityContext:
      runAsUser: 2000
      allowPrivilegeEscalation: false
```

Create the Pod:

```
kubectl apply -f https://k8s.io/examples/pods/security/security-context-2.yaml
```

Verify that the Pod's Container is running:

```
kubectl get pod security-context-demo-2
```

Get a shell into the running Container:

```
kubectl exec -it security-context-demo-2 -- sh
```

In your shell, list the running processes:

```
ps aux
```

The output shows that the processes are running as user 2000. This is the value of `runAsUser` specified for the Container. It overrides the value 1000 that is specified for the Pod.

USER	PID	%CPU	%MEM	VSZ	RSS	TTY	STAT	START	TIME	COMMAND
2000	1	0.0	0.0	4336	764	?	Ss	20:36	0:00	/bin/sh
2000	8	0.1	0.5	772124	22604	?	S1	20:36	0:00	node s
...										

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
2000	8	0.1	0.5	772124	22604	?	S1	20:36	0:00	node s
...										

Exit your shell:

```
exit
```

Set capabilities for a Container

With [Linux capabilities](#), you can grant certain privileges to a process without granting all the privileges of the root user. To add or remove Linux capabilities for a Container, include the `capabilities` field in the `securityContext` section of the Container manifest.

First, see what happens when you don't include a `capabilities` field. Here is configuration file that does not add or remove any Container capabilities:

```
pods/security/security-context-3.yaml
```

```
apiVersion: v1
kind: Pod
metadata:
  name: security-context-demo-3
spec:
  containers:
    - name: sec-ctx-3
      image: gcr.io/google-samples/node-hello:1.0
```

Create the Pod:

```
kubectl apply -f https://k8s.io/examples/pods/security/security-context-3.yaml
```

Verify that the Pod's Container is running:

```
kubectl get pod security-context-demo-3
```

Get a shell into the running Container:

```
kubectl exec -it security-context-demo-3 -- sh
```

In your shell, list the running processes:

```
ps aux
```

The output shows the process IDs (PIDs) for the Container:

USER	PID	%CPU	%MEM	VSZ	RSS	TTY	STAT	START	TIME	COMMAND
root	1	0.0	0.0	4336	796	?	Ss	18:17	0:00	/bin/sh -c nod
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:

[pods/security/security-context-4.yaml](#) 

```
apiVersion: v1  
kind: Pod  
metadata:  
  name: security-context-demo-4  
spec:  
  containers:  
    - name: sec-ctx-4  
      image: gcr.io/google-samples/node-hello:1.0  
      securityContext:  
        capabilities:  
          add: ["NET_ADMIN", "SYS_TIME"]
```

Create the Pod:

```
kubectl apply -f https://k8s.io/examples/pods/security/security-contex
```

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.

Set the Seccomp Profile for a Container

To set the Seccomp profile for a Container, include the `seccompProfile` field in the `securityContext` section of your Pod or Container manifest. The `seccompProfile` field is a [SeccompProfile](#) object consisting of `type` and `localhostProfile`. Valid options for `type` include `RuntimeDefault`, `Unconfined`, and `Localhost`. `localhostProfile` must only be set if `type: Localhost`. It indicates the path of the pre-configured profile on the node, relative to the kubelet's configured Seccomp profile location (configured with the `--root-dir` flag).

Here is an example that sets the Seccomp profile to the node's container runtime default profile:

```
...
  securityContext:
    seccompProfile:
      type: RuntimeDefault
```

Here is an example that sets the Seccomp profile to a pre-configured file at `<kubelet-root-dir>/seccomp/my-profiles/profile-allow.json`:

```
...
  securityContext:
    seccompProfile:
      type: Localhost
      localhostProfile: my-profiles/profile-allow.json
```

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.

Efficient SELinux volume relabeling

FEATURE STATE: [Kubernetes v1.27 \[beta\]](#)

By default, the container runtime recursively assigns SELinux label to all files on all Pod volumes. To speed up this process, Kubernetes can change the SELinux label of a volume instantly by using a mount option `-o context=<label>`.

To benefit from this speedup, all these conditions must be met:

- The [feature gates](#) `ReadWriteOncePod` and `SELinuxMountReadWriteOncePod` must be enabled.
- Pod must use `PersistentVolumeClaim` with `accessModes`:
`["ReadWriteOncePod"]`.
- Pod (or all its Containers that use the `PersistentVolumeClaim`) must have `seLinuxOptions` set.
- The corresponding `PersistentVolume` must be either:
 - A volume that uses the legacy in-tree `iscsi`, `rbd` or `fc` volume type.
 - Or a volume that uses a [CSI](#) driver. The CSI driver must announce that it supports mounting with `-o context` by setting `spec.seLinuxMount: true` in its `CSIDriver` instance.

For any other volume types, SELinux relabelling happens another way: the container runtime recursively changes the SELinux label for all inodes (files and directories) in the volume. The more files and directories in the volume, the longer that relabelling takes.

Note:

If you are running Kubernetes v1.25, refer to the v1.25 version of this task page: [Configure a Security Context for a Pod or Container](#) (v1.25). There is an important note in that documentation about a situation where the kubelet can lose track of volume labels after restart. This deficiency has been fixed in Kubernetes 1.26.

Discussion

The security context for a Pod applies to the Pod's Containers and also to the Pod's Volumes when applicable. Specifically `fsGroup` and `seLinuxOptions` are applied to Volumes as follows:

- `fsGroup` : Volumes that support ownership management are modified to be owned and writable by the GID specified in `fsGroup`. See the [Ownership Management design document](#) for more details.
- `seLinuxOptions` : Volumes that support SELinux labeling are relabeled to be accessible by the label specified under `seLinuxOptions`. Usually you only need to set the `level` section. This sets the [Multi-Category Security \(MCS\)](#) label given to all Containers in the Pod as well as the Volumes.

Warning: After you specify an MCS label for a Pod, all Pods with the same label can access the Volume. If you need inter-Pod protection, you must assign a unique MCS label to each Pod.

Clean up

Delete the Pod:

```
kubectl delete pod security-context-demo
kubectl delete pod security-context-demo-2
kubectl delete pod security-context-demo-3
kubectl delete pod security-context-demo-4
```

What's next

- [PodSecurityContext](#)
- [SecurityContext](#)
- [CRI Plugin Config Guide](#)
- [Security Contexts design document](#)
- [Ownership Management design document](#)
- [PodSecurity Admission](#)
- [AllowPrivilegeEscalation design document](#)
- For more information about security mechanisms in Linux, see [Overview of Linux Kernel Security Features](#) (Note: Some information is out of date)

3.13 - Configure Service Accounts for Pods

Kubernetes offers two distinct ways for clients that run within your cluster, or that otherwise have a relationship to your cluster's control plane to authenticate to the API server.

A *service account* provides an identity for processes that run in a Pod, and maps to a ServiceAccount object. When you authenticate to the API server, you identify yourself as a particular *user*. Kubernetes recognises the concept of a user, however, Kubernetes itself does **not** have a User API.

This task guide is about ServiceAccounts, which do exist in the Kubernetes API. The guide shows you some ways to configure ServiceAccounts 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. It is recommended to run this tutorial on a cluster with at least two nodes that are not acting as control plane hosts. If you do not already have a cluster, you can create one by using [minikube](#) or you can use one of these Kubernetes playgrounds:

- [Killercoda](#)
- [Play with Kubernetes](#)

Use the default service account to access the API server

When Pods contact the API server, Pods authenticate as a particular ServiceAccount (for example, `default`). There is always at least one ServiceAccount in each namespace.

Every Kubernetes namespace contains at least one ServiceAccount: the default ServiceAccount for that namespace, named `default`. If you do not specify a ServiceAccount when you create a Pod, Kubernetes automatically assigns the ServiceAccount named `default` in that namespace.

You can fetch the details for a Pod you have created. For example:

```
kubectl get pods/<podname> -o yaml
```

In the output, you see a field `spec.serviceAccountName`. Kubernetes automatically sets that value if you don't specify it when you create a Pod.

An application running inside a Pod can access the Kubernetes API using automatically mounted service account credentials. See [accessing the Cluster](#) to learn more.

When a Pod authenticates as a ServiceAccount, its level of access depends on the [authorization plugin and policy](#) in use.

Opt out of API credential automounting

If you don't want the `kubelet` to automatically mount a ServiceAccount's API credentials, you can opt out of the default behavior. You can opt out of automounting API credentials on `/var/run/secrets/kubernetes.io/serviceaccount/token` for a service account by setting `automountServiceAccountToken: false` on the ServiceAccount:

For example:

```
apiVersion: v1
kind: ServiceAccount
metadata:
  name: build-robot
automountServiceAccountToken: false
...
```

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
...
```

If both the ServiceAccount and the Pod's `.spec` specify a value for `automountServiceAccountToken`, the Pod spec takes precedence.

Use more than one ServiceAccount

Every namespace has at least one ServiceAccount: the default ServiceAccount resource, called `default`. You can list all ServiceAccount resources in your [current namespace](#) with:

```
kubectl get serviceaccounts
```

The output is similar to this:

NAME	SECRETS	AGE
default	1	1d

You can create additional ServiceAccount objects like this:

```
kubectl apply -f - <<EOF
apiVersion: v1
kind: ServiceAccount
metadata:
  name: build-robot
EOF
```

The name of a ServiceAccount object must be a valid [DNS subdomain name](#).

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

```
kubectl get serviceaccounts/build-robot -o yaml
```

The output is similar to this:

```
apiVersion: v1
kind: ServiceAccount
metadata:
  creationTimestamp: 2019-06-16T00:12:34Z
  name: build-robot
  namespace: default
  resourceVersion: "272500"
  uid: 721ab723-13bc-11e5-aec2-42010af0021e
```

You can use authorization plugins to [set permissions on service accounts](#).

To use a non-default service account, set the `spec.serviceAccountName` field of a Pod to the name of the ServiceAccount you wish to use.

You can only set the `serviceAccountName` field when creating a Pod, or in a template for a new Pod. You cannot update the `.spec.serviceAccountName` field of a Pod that already exists.

Note: The `.spec.serviceAccount` field is a deprecated alias for `.spec.serviceAccountName`. If you want to remove the fields from a workload resource, set both fields to empty explicitly on the [pod template](#).

Cleanup

If you tried creating `build-robot` ServiceAccount from the example above, you can clean it up by running:

```
kubectl delete serviceaccount/build-robot
```

Manually create an API token for a ServiceAccount

Suppose you have an existing service account named "build-robot" as mentioned earlier.

You can get a time-limited API token for that ServiceAccount using `kubectl`:

```
kubectl create token build-robot
```

The output from that command is a token that you can use to authenticate as that ServiceAccount. You can request a specific token duration using the `--duration` command line argument to `kubectl create token` (the actual

duration of the issued token might be shorter, or could even be longer).

Note:

Versions of Kubernetes before v1.22 automatically created long term credentials for accessing the Kubernetes API. This older mechanism was based on creating token Secrets that could then be mounted into running Pods. In more recent versions, including Kubernetes v1.28, API credentials are obtained directly by using the [TokenRequest](#) API, and are mounted into Pods using a [projected volume](#). The tokens obtained using this method have bounded lifetimes, and are automatically invalidated when the Pod they are mounted into is deleted.

You can still manually create a service account token Secret; for example, if you need a token that never expires. However, using the [TokenRequest](#) subresource to obtain a token to access the API is recommended instead.

Manually create a long-lived API token for a ServiceAccount

If you want to obtain an API token for a ServiceAccount, you create a new Secret with a special annotation, `kubernetes.io/service-account.name`.

```
kubectl apply -f - <<EOF
apiVersion: v1
kind: Secret
metadata:
  name: build-robot-secret
  annotations:
    kubernetes.io/service-account.name: build-robot
type: kubernetes.io/service-account-token
EOF
```

If you view the Secret using:

```
kubectl get secret/build-robot-secret -o yaml
```

you can see that the Secret now contains an API token for the "build-robot" ServiceAccount.

Because of the annotation you set, the control plane automatically generates a token for that ServiceAccounts, and stores them into the associated Secret. The control plane also cleans up tokens for deleted ServiceAccounts.

```
kubectl describe secrets/build-robot-secret
```

The output is similar to this:

```
Name: build-robot-secret
Namespace: default
Labels: <none>
Annotations: kubernetes.io/service-account.name: build-robot
              kubernetes.io/service-account.uid: da68f9c6-9d26-11e7-b
Type: kubernetes.io/service-account-token

Data
=====
ca.crt: 1338 bytes
namespace: 7 bytes
token: ...

◀ ▶
```

Note:

The content of `token` is elided here.

Take care not to display the contents of a `kubernetes.io/service-account-token` Secret somewhere that your terminal / computer screen could be seen by an onlooker.

When you delete a ServiceAccount that has an associated Secret, the Kubernetes control plane automatically cleans up the long-lived token from that Secret.

Note:

If you view the ServiceAccount using:

```
kubectl get serviceaccount build-robot -o yaml
```

You can't see the `build-robot-secret` Secret in the ServiceAccount API objects `.secrets` field because that field is only populated with auto-generated Secrets.

Add ImagePullSecrets to a service account

First, [create an imagePullSecret](#). Next, verify it has been created. For example:

- Create an imagePullSecret, as described in [Specifying ImagePullSecrets on a Pod](#).

```
kubectl create secret docker-registry myregistrykey --docker-server=docker://localhost --docker-username=DUMMY_USERNAME --docker-password=DUMMY_PASSWORD --docker-email=DUMMY_DOCKER_EMAIL
```

- Verify it has been created.

```
kubectl get secrets myregistrykey
```

The output is similar to this:

NAME	TYPE	DATA	AGE
myregistrykey	kubernetes.io/.dockerconfigjson	1	1d

Add image pull secret to service account

Next, modify the default service account for the namespace to use this Secret as an imagePullSecret.

```
kubectl patch serviceaccount default -p '{"imagePullSecrets": [{"name": "myregistrykey"}]}
```

You can achieve the same outcome by editing the object manually:

```
kubectl edit serviceaccount/default
```

The output of the `sa.yaml` file is similar to this:

Your selected text editor will open with a configuration looking something like this:

```
apiVersion: v1
kind: ServiceAccount
metadata:
  creationTimestamp: 2021-07-07T22:02:39Z
  name: default
  namespace: default
  resourceVersion: "243024"
  uid: 052fb0f4-3d50-11e5-b066-42010af0d7b6
```

Using your editor, delete the line with key `resourceVersion`, add lines for `imagePullSecrets:` and save it. Leave the `uid` value set the same as you found it.

After you made those changes, the edited ServiceAccount looks something like this:

```
apiVersion: v1
kind: ServiceAccount
metadata:
  creationTimestamp: 2021-07-07T22:02:39Z
  name: default
  namespace: default
  uid: 052fb0f4-3d50-11e5-b066-42010af0d7b6
imagePullSecrets:
- name: myregistrykey
```

Verify that imagePullSecrets are set for new Pods

Now, when a new Pod is created in the current namespace and using the default ServiceAccount, the new Pod has its `spec.imagePullSecrets` field set automatically:

```
kubectl run nginx --image=nginx --restart=Never  
kubectl get pod nginx -o=jsonpath='{.spec.imagePullSecrets[0].name}{"\\"}'
```

The output is:

```
myregistrykey
```

ServiceAccount token volume projection

FEATURE STATE: [Kubernetes v1.20 \[stable\]](#)

Note:

To enable and use token request projection, you must specify each of the following command line arguments to `kube-apiserver`:

`--service-account-issuer`

defines the Identifier of the service account token issuer. You can specify the `--service-account-issuer` argument multiple times, this can be useful to enable a non-disruptive change of the issuer. When this flag is specified multiple times, the first is used to generate tokens and all are used to determine which issuers are accepted. You must be running Kubernetes v1.22 or later to be able to specify `--service-account-issuer` multiple times.

`--service-account-key-file`

specifies the path to a file containing PEM-encoded X.509 private or public keys (RSA or ECDSA), used to verify ServiceAccount tokens. The specified file can contain multiple keys, and the flag can be specified multiple times with different files. If specified multiple times, tokens signed by any of the specified keys are considered valid by the Kubernetes API server.

`--service-account-signing-key-file`

specifies the path to a file that contains the current private key of the service account token issuer. The issuer signs issued ID tokens with this private key.

`--api-audiences` (can be omitted)

defines audiences for ServiceAccount tokens. The service account token authenticator validates that tokens used against the API are bound to at least one of these audiences. If `api-audiences` is specified multiple times, tokens for any of the specified audiences are considered valid by the Kubernetes API server. If you specify the `--service-account-issuer` command line argument but you don't set `--api-audiences`, the control plane defaults to a single element audience list that contains only the issuer URL.

The kubelet can also project a ServiceAccount token into a Pod. You can specify desired properties of the token, such as the audience and the validity duration. These properties are *not* configurable on the default ServiceAccount token. The token will also become invalid against the API when either the Pod or the ServiceAccount is deleted.

You can configure this behavior for the `spec` of a Pod using a [projected volume](#) type called `ServiceAccountToken`.

Launch a Pod using service account token projection

To provide a Pod with a token with an audience of `vault` and a validity duration of two hours, you could define a Pod manifest that is similar to:

[pods/pod-projected-svc-token.yaml](#) 

```
apiVersion: v1
kind: Pod
metadata:
  name: nginx
spec:
  containers:
    - image: nginx
      name: nginx
    volumeMounts:
      - mountPath: /var/run/secrets/tokens
        name: vault-token
  serviceAccountName: build-robot
  volumes:
    - name: vault-token
  projected:
    sources:
      - serviceAccountToken:
          path: vault-token
          expirationSeconds: 7200
          audience: vault
```

Create the Pod:

```
kubectl create -f https://k8s.io/examples/pods/pod-projected-svc-token
```

The kubelet will: request and store the token on behalf of the Pod; make the token available to the Pod at a configurable file path; and refresh the token as it approaches expiration. The kubelet proactively requests rotation for the token if it is older than 80% of its total time-to-live (TTL), or if the token is older than 24 hours.

The application is responsible for reloading the token when it rotates. It's often good enough for the application to load the token on a schedule (for example: once every 5 minutes), without tracking the actual expiry time.

Service account issuer discovery

FEATURE STATE: [Kubernetes v1.21 \[stable\]](#)

If you have enabled [token projection](#) for ServiceAccounts in your cluster, then you can also make use of the discovery feature. Kubernetes provides a way for clients to federate as an *identity provider*, so that one or more external systems can act as a *relying party*.

Note:

The issuer URL must comply with the [OIDC Discovery Spec](#). In practice, this means it must use the `https` scheme, and should serve an OpenID provider configuration at `{service-account-issuer}/.well-known/openid-configuration`.

If the URL does not comply, ServiceAccount issuer discovery endpoints are not registered or accessible.

When enabled, the Kubernetes API server publishes an OpenID Provider Configuration document via HTTP. The configuration document is published at `/.well-known/openid-configuration`. The OpenID Provider Configuration is sometimes referred to as the *discovery document*. The Kubernetes API server publishes the related JSON Web Key Set (JWKS), also via HTTP, at `/openid/v1/jwks`.

Note: The responses served at `/.well-known/openid-configuration` and `/openid/v1/jwks` are designed to be OIDC compatible, but not strictly OIDC compliant. Those documents contain only the parameters necessary to perform validation of Kubernetes service account tokens.

Clusters that use RBAC include a default ClusterRole called `system:service-account-issuer-discovery`. A default ClusterRoleBinding assigns this role to the `system:serviceaccounts` group, which all ServiceAccounts implicitly belong to. This allows pods running on the cluster to access the service account discovery document via their mounted service account token. Administrators may, additionally, choose to bind the role to `system:authenticated` or `system:unauthenticated` depending on their security requirements and which external systems they intend to federate with.

The JWKS response contains public keys that a relying party can use to validate the Kubernetes service account tokens. Relying parties first query for the OpenID Provider Configuration, and use the `jwks_uri` field in the response to find the JWKS.

In many cases, Kubernetes API servers are not available on the public internet, but public endpoints that serve cached responses from the API server can be made available by users or by service providers. In these cases, it is possible to override the `jwks_uri` in the OpenID Provider Configuration so that it points to the public endpoint, rather than the API server's address, by passing the `--service-account-jwks-uri` flag to the API server. Like the issuer URL, the JWKS URI is required to use the `https` scheme.

What's next

See also:

- Read the [Cluster Admin Guide to Service Accounts](#)
- Read about [Authorization in Kubernetes](#)
- Read about [Secrets](#)
 - or learn to [distribute credentials securely using Secrets](#)
 - but also bear in mind that using Secrets for authenticating as a ServiceAccount is deprecated. The recommended alternative is [ServiceAccount token volume projection](#).
- Read about [projected volumes](#).
- For background on OIDC discovery, read the [ServiceAccount signing key retrieval](#) Kubernetes Enhancement Proposal
- Read the [OIDC Discovery Spec](#)

3.14 - 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 container image registry or repository. There are many private registries in use. This task uses [Docker Hub](#) as an example registry.

ⓘ This item links to a third party project or product that is not part of Kubernetes itself. [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. It is recommended to run this tutorial on a cluster with at least two nodes that are not acting as control plane hosts. If you do not already have a cluster, you can create one by using [minikube](#) or you can use one of these Kubernetes playgrounds:
 - [Killercoda](#)
 - [Play with Kubernetes](#)
- To do this exercise, you need the `docker` command line tool, and a [Docker ID](#) for which you know the password.
- If you are using a different private container registry, you need the command line tool for that registry and any login information for the registry.

Log in to Docker Hub

On your laptop, you must authenticate with a registry in order to pull a private image.

Use the `docker` tool to log in to Docker Hub. See the *log in* section of [Docker ID accounts](#) for more information.

```
docker login
```

When prompted, enter your Docker ID, and then the credential you want to use (access token, or the password for your Docker ID).

The login process creates or updates a `config.json` file that holds an authorization token. Review [how Kubernetes interprets this file](#).

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. In that case, you can create a secret directly. See [Create a Secret by providing credentials on the command line](#).

Create a Secret based on existing credentials

A Kubernetes cluster uses the Secret of `kubernetes.io/dockerconfigjson` type to authenticate with a container registry to pull a private image.

If you already ran `docker login`, you can copy that credential into Kubernetes:

```
kubectl create secret generic regcred \
    --from-file=.dockerconfigjson=<path/to/.docker/config.json> \
    --type=kubernetes.io/dockerconfigjson
```

If you need more control (for example, to set a namespace or a label on the new secret) then you can customise the Secret before storing it. Be sure to:

- set the name of the data item to `.dockerconfigjson`
 - base64 encode the Docker configuration file and then paste that string, unbroken as the value for field `data[".dockerconfigjson"]`
 - set `type` to `kubernetes.io/dockerconfigjson`

Example:

If you get the error message `error: no objects passed to create`, it may mean the base64 encoded string is invalid. If you get an error message like `Secret "myregistrykey" is invalid: data[.dockerconfigjson]: invalid value` ..., it means the base64 encoded string in the data was successfully decoded, but could not be parsed as a `.docker/config.json` file.

Create a Secret by providing credentials on the command line

Create this Secret, naming it `regcred`:

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

where:

- `<your-registry-server>` is your Private Docker Registry FQDN. Use <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`.

Note: Typing secrets on the command line may store them in your shell history unprotected, and those secrets might also be visible to other users on your PC during the time that `kubectl` is running.

Inspecting the Secret `regcred`

To understand the contents of the `regcred` Secret you created, start by viewing the Secret in YAML format:

```
kubectl get secret regcred --output=yaml
```

The output is similar to this:

```
apiVersion: v1
kind: Secret
metadata:
...
name: regcred
...
data:
.dockerconfigjson: eyJodHRwczovL2luZGV4L ... J0QUl6RTIifX0=
type: kubernetes.io/dockerconfigjson
```

The value of the `.dockerconfigjson` field is a base64 representation of your Docker credentials.

To understand what is in the `.dockerconfigjson` field, convert the secret data to a readable format:

```
kubectl get secret regcred --output="jsonpath={.data.\.dockerconfigjso
```

The output is similar to this:

```
{"auths":{"your.private.registry.example.com":{"username":"janedoe","password": "janedoe1234567890!@#%^&*()?"}}
```

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

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

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

janedoe:xxxxxxxxxxxx

Notice that the Secret data contains the authorization token similar to your local `~/.docker/config.json` file.

You have successfully set your Docker credentials as a Secret called `regcred` in the cluster.

Create a Pod that uses your Secret

Here is a manifest for an example Pod that needs access to your Docker credentials in `regcred`:

[pods/private-reg-pod.yaml](#)

```
apiVersion: v1
kind: Pod
metadata:
  name: private-reg
spec:
  containers:
    - name: private-reg-container
      image: <your-private-image>
  imagePullSecrets:
    - name: regcred
```

Download the above file onto your computer.

```
curl -L -o my-private-reg-pod.yaml https://k8s.io/examples/pods/private
```

In file `my-private-reg-pod.yaml`, replace `<your-private-image>` with the path to an image in a private registry such as:

`your.private.registry.example.com/janedoe/jdoe-private:v1`

To pull the image from the private registry, Kubernetes needs credentials. The `imagePullSecrets` field in the configuration file specifies that Kubernetes should get the credentials from a Secret named `regcred`.

Create a Pod that uses your Secret, and verify that the Pod is running:

```
kubectl apply -f my-private-reg-pod.yaml  
kubectl get pod private-reg
```

Note: To use image pull secrets for a Pod (or a Deployment, or other object that has a pod template that you are using), you need to make sure that the appropriate Secret does exist in the right namespace. The namespace to use is the same namespace where you defined the Pod.

Also, in case the Pod fails to start with the status `ImagePullBackOff`, view the Pod events:

```
kubectl describe pod private-reg
```

If you then see an event with the reason set to `FailedToRetrieveImagePullSecret`, Kubernetes can't find a Secret with name (`regcred`, in this example). If you specify that a Pod needs image pull credentials, the kubelet checks that it can access that Secret before attempting to pull the image.

Make sure that the Secret you have specified exists, and that its name is spelled properly.

```
Events:  
... Reason ... Message  
-----  
... FailedToRetrieveImagePullSecret ... Unable to retrieve some i
```

What's next

- Learn more about [Secrets](#)
 - or read the API reference for [Secret](#)
- Learn more about [using a private registry](#).
- Learn more about [adding image pull secrets to a service account](#).
- See [kubectl create secret docker-registry](#).
- See the `imagePullSecrets` field within the [container definitions](#) of a Pod

3.15 - Configure Liveness, Readiness and Startup Probes

This page shows how to configure liveness, readiness and startup probes for containers.

The [kubelet](#) uses liveness probes to know when to restart a container. For example, liveness probes could catch a deadlock, where an application is running, but unable to make progress. Restarting a container in such a state can help to make the application more available despite bugs.

A common pattern for liveness probes is to use the same low-cost HTTP endpoint as for readiness probes, but with a higher failureThreshold. This ensures that the pod is observed as not-ready for some period of time before it is hard killed.

The kubelet uses readiness probes to know when a container is ready to start accepting traffic. A Pod is considered ready when all of its containers are ready. One use of this signal is to control which Pods are used as backends for Services. When a Pod is not ready, it is removed from Service load balancers.

The kubelet uses startup probes to know when a container application has started. If such a probe is configured, liveness and readiness probes do not start until it succeeds, making sure those probes don't interfere with the application startup. This can be used to adopt liveness checks on slow starting containers, avoiding them getting killed by the kubelet before they are up and running.

Caution: Liveness probes can be a powerful way to recover from application failures, but they should be used with caution. Liveness probes must be configured carefully to ensure that they truly indicate unrecoverable application failure, for example a deadlock.

Note: Incorrect implementation of liveness probes can lead to cascading failures. This results in restarting of container under high load; failed client requests as your application became less scalable; and increased workload on remaining pods due to some failed pods. Understand the difference between readiness and liveness probes and when to apply them for your app.

Before you begin

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

- [Killercoda](#)
- [Play with Kubernetes](#)

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 `registry.k8s.io/busybox` image. Here is the configuration file for the Pod:

```
pods/probe/exec-liveness.yaml 
```

```
apiVersion: v1
kind: Pod
metadata:
  labels:
    test: liveness
  name: liveness-exec
spec:
  containers:
  - name: liveness
    image: registry.k8s.io/busybox
    args:
    - /bin/sh
    - -c
    - touch /tmp/healthy; sleep 30; rm -f /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 seconds before performing the first probe. To perform a probe, the kubelet executes the command `cat /tmp/healthy` in the target 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 -f /tmp/healthy; sleep 600"
```

For the first 30 seconds of the container's life, there is a `/tmp/healthy` file. So during the first 30 seconds, the command `cat /tmp/healthy` returns a success code. After 30 seconds, `cat /tmp/healthy` returns a failure code.

Create the Pod:

```
kubectl apply -f https://k8s.io/examples/pods/probe/exec-liveness.yaml
```

Within 30 seconds, view the Pod events:

```
kubectl describe pod liveness-exec
```

The output indicates that no liveness probes have failed yet:

Type	Reason	Age	From	Message
Normal	Scheduled	11s	default-scheduler	Successfully assigned default-pod to node01
Normal	Pulling	9s	kubelet, node01	Pulling image "registry.k8s.io/liveness:latest"
Normal	Pulled	7s	kubelet, node01	Successfully pulled image 'registry.k8s.io/liveness:latest'
Normal	Created	7s	kubelet, node01	Created container liveness
Normal	Started	7s	kubelet, node01	Started container liveness

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 failed containers have been killed and recreated.

Type	Reason	Age	From	Message
Normal	Scheduled	57s	default-scheduler	Successfully assigned default-pod to node01
Normal	Pulling	55s	kubelet, node01	Pulling image "registry.k8s.io/liveness:latest"
Normal	Pulled	53s	kubelet, node01	Successfully pulled image 'registry.k8s.io/liveness:latest'
Normal	Created	53s	kubelet, node01	Created container liveness
Normal	Started	53s	kubelet, node01	Started container liveness
Warning	Unhealthy	10s (x3 over 20s)	kubelet, node01	Liveness probe failed: http://liveness:8080/liveness
Normal	Killing	10s	kubelet, node01	Container will be killed

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. Note that the `RESTARTS` counter increments as soon as a failed container comes back to the running state:

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 `registry.k8s.io/liveness` image.

[pods/probe/http-liveness.yaml](#) 

```

apiVersion: v1
kind: Pod
metadata:
  labels:
    test: liveness
  name: liveness-http
spec:
  containers:
    - name: liveness
      image: registry.k8s.io/liveness
      args:
        - /server
      livenessProbe:
        httpGet:
          path: /healthz
          port: 8080
          httpHeaders:
            - name: 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 apply -f https://k8s.io/examples/pods/probe/http-liveness.yaml
```

After 10 seconds, view Pod events to verify that liveness probes have failed and the container has been restarted:

```
kubectl describe pod liveness-http
```

In releases after v1.13, local HTTP proxy environment variable settings do not affect the HTTP liveness probe.

Define a TCP liveness probe

A third type of liveness probe uses a TCP socket. With this configuration, the kubelet will attempt to open a socket to your container on the specified port. If it can establish a connection, the container is considered healthy, if it can't it is considered a failure.

[pods/probe/tcp-liveness-readiness.yaml](#) 

```
apiVersion: v1
kind: Pod
metadata:
  name: goproxy
  labels:
    app: goproxy
spec:
  containers:
  - name: goproxy
    image: registry.k8s.io/goproxy:0.1
    ports:
    - containerPort: 8080
  readinessProbe:
    tcpSocket:
      port: 8080
    initialDelaySeconds: 15
    periodSeconds: 10
  livenessProbe:
    tcpSocket:
      port: 8080
    initialDelaySeconds: 15
    periodSeconds: 10
```

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 15 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. Similar to the readiness probe, this will attempt to connect to the `goproxy` container on port 8080. If the liveness probe fails, the container will be restarted.

To try the TCP liveness check, create a Pod:

```
kubectl apply -f https://k8s.io/examples/pods/probe/tcp-liveness-readiness-probe.yaml
```

After 15 seconds, view Pod events to verify that liveness probes:

```
kubectl describe pod goproxy
```

Define a gRPC liveness probe

FEATURE STATE: [Kubernetes v1.27 \[stable\]](#)

If your application implements the [gRPC Health Checking Protocol](#), this example shows how to configure Kubernetes to use it for application liveness checks. Similarly you can configure readiness and startup probes.

Here is an example manifest:

```
pod/probe/grpc-liveness.yaml
```

```
apiVersion: v1
kind: Pod
metadata:
  name: etcd-with-grpc
spec:
  containers:
  - name: etcd
    image: registry.k8s.io/etcd:3.5.1-0
    command: [ "/usr/local/bin/etcd", "--data-dir", "/var/lib/etcd",
    ports:
    - containerPort: 2379
    livenessProbe:
      grpc:
        port: 2379
      initialDelaySeconds: 10
```

To use a gRPC probe, `port` must be configured. If you want to distinguish probes of different types and probes for different features you can use the `service` field. You can set `service` to the value `liveness` and make your gRPC Health Checking endpoint respond to this request differently than when you set `service` set to `readiness`. This lets you use the same endpoint for different kinds of container health check rather than listening on two different ports. If you want to specify your own custom service name and also specify a probe type, the Kubernetes project recommends that you use a name that concatenates those. For example: `myservice-liveness` (using `-` as a separator).

Note: Unlike HTTP or TCP probes, you cannot specify the health check port by name, and you cannot configure a custom hostname.

Configuration problems (for example: incorrect port or service, unimplemented health checking protocol) are considered a probe failure, similar to HTTP and TCP probes.

To try the gRPC liveness check, create a Pod using the command below. In the example below, the etcd pod is configured to use gRPC liveness probe.

```
kubectl apply -f https://k8s.io/examples/pods/probe/grpc-liveness.yaml
```

After 15 seconds, view Pod events to verify that the liveness check has not failed:

```
kubectl describe pod etcd-with-grpc
```

When using a gRPC probe, there are some technical details to be aware of:

- The probes run against the pod IP address or its hostname. Be sure to configure your gRPC endpoint to listen on the Pod's IP address.
- The probes do not support any authentication parameters (like `-tls`).
- There are no error codes for built-in probes. All errors are considered as probe failures.
- If `ExecProbeTimeout` feature gate is set to `false`, `grpc-health-probe` does **not** respect the `timeoutSeconds` setting (which defaults to 1s), while built-in probe would fail on timeout.

Use a named port

You can use a named [port](#) for HTTP and TCP probes. gRPC probes do not support named ports.

For example:

```
ports:  
- name: liveness-port  
  containerPort: 8080  
  hostPort: 8080  
  
livenessProbe:  
  httpGet:  
    path: /healthz  
    port: liveness-port
```

Protect slow starting containers with startup probes

Sometimes, you have to deal with legacy applications that might require an additional startup time on their first initialization. In such cases, it can be tricky to set up liveness probe parameters without compromising the fast response to deadlocks that motivated such a probe. The trick is to set up a startup probe with the same command, HTTP or TCP check, with a `failureThreshold * periodSeconds` long enough to cover the worst case startup time.

So, the previous example would become:

```

ports:
- name: liveness-port
  containerPort: 8080
  hostPort: 8080

livenessProbe:
  httpGet:
    path: /healthz
    port: liveness-port
  failureThreshold: 1
  periodSeconds: 10

startupProbe:
  httpGet:
    path: /healthz
    port: liveness-port
  failureThreshold: 30
  periodSeconds: 10

```

Thanks to the startup probe, the application will have a maximum of 5 minutes ($30 * 10 = 300$ s) to finish its startup. Once the startup probe has succeeded once, the liveness probe takes over to provide a fast response to container deadlocks. If the startup probe never succeeds, the container is killed after 300s and subject to the pod's `restartPolicy`.

Define readiness probes

Sometimes, applications are temporarily unable to serve traffic. For example, an application might need to load large data or configuration files during startup, or depend on external services after startup. In such cases, you don't want to kill the application, but you don't want to send it requests either. Kubernetes provides readiness probes to detect and mitigate these situations. A pod with containers reporting that they are not ready does not receive traffic through Kubernetes Services.

Note: Readiness probes runs on the container during its whole lifecycle.

Caution: Liveness probes *do not* wait for readiness probes to succeed. If you want to wait before executing a liveness probe you should use `initialDelaySeconds` or a `startupProbe`.

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 startup, liveness and readiness checks:

- `initialDelaySeconds` : Number of seconds after the container has started before startup, liveness or readiness probes are initiated. If a startup probe is defined, liveness and readiness probe delays do not begin until the startup probe has succeeded. If the value of `periodSeconds` is greater than `initialDelaySeconds` then the `initialDelaySeconds` would be ignored. Defaults to 0 seconds. Minimum value is 0.
- `periodSeconds` : How often (in seconds) to perform the probe. Default to 10 seconds. The 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 and startup Probes. Minimum value is 1.
- `failureThreshold` : After a probe fails `failureThreshold` times in a row, Kubernetes considers that the overall check has failed: the container is *not* ready/healthy/live. For the case of a startup or liveness probe, if at least `failureThreshold` probes have failed, Kubernetes treats the container as unhealthy and triggers a restart for that specific container. The kubelet honors the setting of `terminationGracePeriodSeconds` for that container. For a failed readiness probe, the kubelet continues running the container that failed checks, and also continues to run more probes; because the check failed, the kubelet sets the `Ready` [condition](#) on the Pod to `false`.
- `terminationGracePeriodSeconds` : configure a grace period for the kubelet to wait between triggering a shut down of the failed container, and then forcing the container runtime to stop that container. The default is to inherit the Pod-level value for `terminationGracePeriodSeconds` (30 seconds if not specified), and the minimum value is 1. See [probe-level terminationGracePeriodSeconds](#) for more detail.

Caution: Incorrect implementation of readiness probes may result in an ever growing number of processes in the container, and resource starvation if this is left unchecked.

HTTP probes

[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. Defaults to "/".
- `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 port and path to perform the check. The kubelet sends the probe to the Pod's IP address, unless the address is overridden by the optional `host` field in `httpGet`. If `scheme` field is set to `HTTPS`, the kubelet sends an HTTPS request skipping the certificate verification. In most scenarios, you do not want to set the `host` field. Here's one scenario where you would set it. Suppose the container listens on 127.0.0.1 and the Pod's `hostNetwork` field is true. Then `host`, under `httpGet`, should be set to 127.0.0.1. If your pod relies on virtual hosts, which is probably the more common case, you should not use `host`, but rather set the `Host` header in `httpHeaders`.

For an HTTP probe, the kubelet sends two request headers in addition to the mandatory `Host` header:

- `User-Agent` : The default value is `kube-probe/1.28`, where `1.28` is the version of the kubelet.
- `Accept` : The default value is `*/*`.

You can override the default headers by defining `httpHeaders` for the probe. For example:

```
livenessProbe:  
  httpGet:  
    httpHeaders:  
      - name: Accept  
        value: application/json  
  
  startupProbe:  
    httpGet:  
      httpHeaders:  
        - name: User-Agent  
          value: MyUserAgent
```

You can also remove these two headers by defining them with an empty value.

```
livenessProbe:  
  httpGet:  
    httpHeaders:  
      - name: Accept  
        value: ""  
  
  startupProbe:  
    httpGet:  
      httpHeaders:  
        - name: User-Agent  
          value: ""
```

TCP probes

For a TCP probe, the kubelet makes the probe connection at the node, not in the Pod, which means that you can not use a service name in the `host` parameter since the kubelet is unable to resolve it.

Probe-level `terminationGracePeriodSeconds`

FEATURE STATE: `Kubernetes v1.28 [stable]`

In 1.25 and above, users can specify a probe-level `terminationGracePeriodSeconds` as part of the probe specification. When both a pod- and probe-level `terminationGracePeriodSeconds` are set, the kubelet will use the probe-level value.

When setting the `terminationGracePeriodSeconds`, please note the following:

- The kubelet always honors the probe-level `terminationGracePeriodSeconds` field if it is present on a Pod.
- If you have existing Pods where the `terminationGracePeriodSeconds` field is set and you no longer wish to use per-probe termination grace periods, you must delete those existing Pods.

For example:

```
spec:  
  terminationGracePeriodSeconds: 3600  # pod-level  
  containers:  
    - name: test  
      image: ...  
  
  ports:  
    - name: liveness-port  
      containerPort: 8080  
      hostPort: 8080  
  
  livenessProbe:  
    httpGet:  
      path: /healthz  
      port: liveness-port  
    failureThreshold: 1  
    periodSeconds: 60  
    # Override pod-level terminationGracePeriodSeconds #  
    terminationGracePeriodSeconds: 60
```

Probe-level `terminationGracePeriodSeconds` cannot be set for readiness probes. It will be rejected by the API server.

What's next

- Learn more about [Container Probes](#).

You can also read the API references for:

- [Pod](#), and specifically:
 - [container\(s\)](#)
 - [probe\(s\)](#)

3.16 - Assign Pods to Nodes

This page shows how to assign a Kubernetes Pod to a particular node in a Kubernetes cluster.

Before you begin

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

- [Killercoda](#)
- [Play with Kubernetes](#)

To check the version, enter `kubectl version`.

Add a label to a node

1. List the nodes in your cluster, along with their labels:

```
kubectl get nodes --show-labels
```

The output is similar to this:

NAME	STATUS	ROLES	AGE	VERSION	LABELS
worker0	Ready	<none>	1d	v1.13.0	...,kubernetes
worker1	Ready	<none>	1d	v1.13.0	...,kubernetes
worker2	Ready	<none>	1d	v1.13.0	...,kubernetes

2. Choose one of your nodes, and add a label to it:

```
kubectl label nodes <your-node-name> disktype=ssd
```

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

3. Verify that your chosen node has a `disktype=ssd` label:

```
kubectl get nodes --show-labels
```

The output is similar to this:

NAME	STATUS	ROLES	AGE	VERSION	LABELS
worker0	Ready	<none>	1d	v1.13.0	...,disktype=
worker1	Ready	<none>	1d	v1.13.0	...,kubernetes
worker2	Ready	<none>	1d	v1.13.0	...,kubernetes

In the preceding output, you can see that the `worker0` node has a `disktype:ssd` label.

Create a pod that gets scheduled to your chosen node

This pod configuration file describes a pod that has a node selector, `disktype: ssd`. This means that the pod will get scheduled on a node that has a `disktype:ssd` label.

[pods/pod-nginx.yaml](#)

```
apiVersion: v1
kind: Pod
metadata:
  name: nginx
  labels:
    env: test
spec:
  containers:
  - name: nginx
    image: nginx
    imagePullPolicy: IfNotPresent
  nodeSelector:
    disktype: ssd
```

1. Use the configuration file to create a pod that will get scheduled on your chosen node:

```
kubectl apply -f https://k8s.io/examples/pods/pod-nginx.yaml
```

2. Verify that the pod is running on your chosen node:

```
kubectl get pods --output=wide
```

The output is similar to this:

NAME	READY	STATUS	RESTARTS	AGE	IP	NODE
nginx	1/1	Running	0	13s	10.200.0.4	worker0

Create a pod that gets scheduled to specific node

You can also schedule a pod to one specific node via setting `nodeName`.

[pods/pod-nginx-specific-node.yaml](#)

```
apiVersion: v1
kind: Pod
metadata:
  name: nginx
spec:
  nodeName: foo-node # schedule pod to specific node
  containers:
  - name: nginx
    image: nginx
    imagePullPolicy: IfNotPresent
```

Use the configuration file to create a pod that will get scheduled on `foo-node` only.

What's next

- Learn more about [labels and selectors](#).
- Learn more about [nodes](#).

3.17 - Assign Pods to Nodes using Node Affinity

This page shows how to assign a Kubernetes Pod to a particular node using Node Affinity in a Kubernetes cluster.

Before you begin

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

- [Killercoda](#)
- [Play with Kubernetes](#)

Your Kubernetes server must be at or later than version v1.10. To check the version, enter `kubectl version`.

Add a label to a node

1. List the nodes in your cluster, along with their labels:

```
kubectl get nodes --show-labels
```

The output is similar to this:

NAME	STATUS	ROLES	AGE	VERSION	LABELS
worker0	Ready	<none>	1d	v1.13.0	...,kubernetes
worker1	Ready	<none>	1d	v1.13.0	...,kubernetes
worker2	Ready	<none>	1d	v1.13.0	...,kubernetes

2. Choose one of your nodes, and add a label to it:

```
kubectl label nodes <your-node-name> disktype=ssd
```

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

3. Verify that your chosen node has a `disktype=ssd` label:

```
kubectl get nodes --show-labels
```

The output is similar to this:

NAME	STATUS	ROLES	AGE	VERSION	LABELS
worker0	Ready	<none>	1d	v1.13.0	...,disktype=ssd
worker1	Ready	<none>	1d	v1.13.0	...,kubernetes.io/os=linux
worker2	Ready	<none>	1d	v1.13.0	...,kubernetes.io/arch=amd64

In the preceding output, you can see that the `worker0` node has a `disktype=ssd` label.

Schedule a Pod using required node affinity

This manifest describes a Pod that has a

`requiredDuringSchedulingIgnoredDuringExecution` node affinity, `disktype:ssd`. This means that the pod will get scheduled only on a node that has a `disktype=ssd` label.

[pods/pod-nginx-required-affinity.yaml](#) 

```

apiVersion: v1
kind: Pod
metadata:
  name: nginx
spec:
  affinity:
    nodeAffinity:
      requiredDuringSchedulingIgnoredDuringExecution:
        nodeSelectorTerms:
        - matchExpressions:
          - key: disktype
            operator: In
            values:
            - ssd
  containers:
  - name: nginx
    image: nginx
    imagePullPolicy: IfNotPresent

```

1. Apply the manifest to create a Pod that is scheduled onto your chosen node:

```
kubectl apply -f https://k8s.io/examples/pods/pod-nginx-required-
```

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

Schedule a Pod using preferred node affinity

This manifest describes a Pod that has a

```
preferredDuringSchedulingIgnoredDuringExecution node affinity, disktype:  
ssd . This means that the pod will prefer a node that has a disktype=ssd  
label.
```

[pods/pod-nginx-preferred-affinity.yaml](#) 

```
apiVersion: v1
kind: Pod
metadata:
  name: nginx
spec:
  affinity:
    nodeAffinity:
      preferredDuringSchedulingIgnoredDuringExecution:
        - weight: 1
          preference:
            matchExpressions:
              - key: disktype
                operator: In
                values:
                  - ssd
  containers:
    - name: nginx
      image: nginx
      imagePullPolicy: IfNotPresent
```

1. Apply the manifest to create a Pod that is scheduled onto your chosen node:

```
kubectl apply -f https://k8s.io/examples/pods/pod-nginx-preferred
```

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	worker

What's next

Learn more about [Node Affinity](#).

3.18 - Configure Pod Initialization

This page shows how to use an Init Container to initialize a Pod before an application Container runs.

Before you begin

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

- [Killercoda](#)
- [Play with Kubernetes](#)

To check the version, enter `kubectl version`.

Create a Pod that has an Init Container

In this exercise you create a Pod that has one application Container and one Init Container. The init container runs to completion before the application container starts.

Here is the configuration file for the Pod:

[pods/init-containers.yaml](#) 

```
apiVersion: v1
kind: Pod
metadata:
  name: init-demo
spec:
  containers:
    - name: nginx
      image: nginx
      ports:
        - containerPort: 80
      volumeMounts:
        - name: workdir
          mountPath: /usr/share/nginx/html
      # These containers are run during pod initialization
  initContainers:
    - name: install
      image: busybox:1.28
      command:
        - wget
        - "-O"
        - "/work-dir/index.html"
        - http://info.cern.ch
      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://info.cern.ch
```

Notice that the init container writes the `index.html` file in the root directory of the nginx server.

Create the Pod:

```
kubectl apply -f https://k8s.io/examples/pods/init-containers.yaml
```

Verify that the nginx container is running:

```
kubectl get pod init-demo
```

The output shows that the nginx container is running:

NAME	READY	STATUS	RESTARTS	AGE
init-demo	1/1	Running	0	1m

Get a shell into the nginx container running in the init-demo Pod:

```
kubectl exec -it init-demo -- /bin/bash
```

In your shell, send a GET request to the nginx server:

```
root@nginx:~# apt-get update
root@nginx:~# apt-get install curl
root@nginx:~# curl localhost
```

The output shows that nginx is serving the web page that was written by the init container:

```
<html><head></head><body><header>
<title>http://info.cern.ch</title>
</header>

<h1>http://info.cern.ch - home of the first website</h1>
...
<li><a href="http://info.cern.ch/hypertext/WWW/TheProject.html">Brow
...

```

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](#)

3.19 - 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. A Container may specify one handler per event.

Before you begin

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

- [KillerCoda](#)
- [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:



```
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/lifecycle-poststart.txt"]
        preStop:
          exec:
            command: ["/bin/sh", "-c", "nginx -s quit; while killall -0 nginx; do sleep 1; done"]
```

In the configuration file, you can see that the postStart command writes a message file to the Container's `/usr/share` directory. The preStop command shuts down nginx gracefully. This is helpful if the Container is being terminated because of a failure.

Create the Pod:

```
kubectl apply -f https://k8s.io/examples/pods/lifecycle-events.yaml
```

Verify that the Container in the Pod is running:

```
kubectl get pod lifecycle-demo
```

Get a shell into the Container running in your Pod:

```
kubectl exec -it lifecycle-demo -- /bin/bash
```

In your shell, verify that the `postStart` handler created the `message` file:

```
root@lifecycle-demo:/# cat /usr/share/message
```

The output shows the text written by the `postStart` handler:

```
Hello from the postStart handler
```

Discussion

Kubernetes sends the `postStart` event immediately after the Container is created. There is no guarantee, however, that the `postStart` handler is called before the Container's entrypoint is called. The `postStart` handler runs asynchronously relative to the Container's code, but Kubernetes' management of the container blocks until the `postStart` handler completes. The Container's status is not set to `RUNNING` until the `postStart` handler completes.

Kubernetes sends the `preStop` event immediately before the Container is terminated. Kubernetes' management of the Container blocks until the `preStop` handler completes, unless the Pod's grace period expires. For more details, see [Pod Lifecycle](#).

Note: Kubernetes only sends the `preStop` event when a Pod or a container in the Pod is *terminated*. This means that the `preStop` hook is not invoked when the Pod is *completed*. About this limitation, please see [Container hooks](#) for the detail.

What's next

- Learn more about [Container lifecycle hooks](#).
- Learn more about the [lifecycle of a Pod](#).

Reference

- [Lifecycle](#)
- [Container](#)
- See `terminationGracePeriodSeconds` in [PodSpec](#)

3.20 - Configure a Pod to Use a ConfigMap

Many applications rely on configuration which is used during either application initialization or runtime. Most times, there is a requirement to adjust values assigned to configuration parameters. ConfigMaps are a Kubernetes mechanism that let you inject configuration data into application pods.

The ConfigMap concept allow you to decouple configuration artifacts from image content to keep containerized applications portable. For example, you can download and run the same container image to spin up containers for the purposes of local development, system test, or running a live end-user workload.

This page provides a series of usage examples demonstrating how to create ConfigMaps and configure Pods using data stored in ConfigMaps.

Before you begin

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

- [Killercoda](#)
- [Play with Kubernetes](#)

You need to have the `wget` tool installed. If you have a different tool such as `curl`, and you do not have `wget`, you will need to adapt the step that downloads example data.

Create a ConfigMap

You can use either `kubectl create configmap` or a ConfigMap generator in `kustomization.yaml` to create a ConfigMap.

Create a ConfigMap using `kubectl create configmap`

Use the `kubectl create configmap` command to create ConfigMaps from [directories](#), [files](#), or [literal values](#):

```
kubectl create configmap <map-name> <data-source>
```

where `<map-name>` is the name you want to assign to the ConfigMap and `<data-source>` is the directory, file, or literal value to draw the data from. The name of a ConfigMap object must be a valid [DNS subdomain name](#).

When you are creating a ConfigMap based on a file, the key in the `<data-source>` defaults to the basename of the file, and the value defaults to the file content.

You can use `kubectl describe` or `kubectl get` to retrieve information about a ConfigMap.

Create a ConfigMap from a directory

You can use `kubectl create configmap` to create a ConfigMap from multiple files in the same directory. When you are creating a ConfigMap based on a directory, `kubectl` identifies files whose filename is a valid key in the directory and packages each of those files into the new ConfigMap. Any directory entries except regular files are ignored (for example: subdirectories, symlinks, devices, pipes, and more).

Note:

Each filename being used for ConfigMap creation must consist of only acceptable characters, which are: letters (A to Z and a to z), digits (0 to 9), '-', '_', or '!'. If you use `kubectl create configmap` with a directory where any of the file names contains an unacceptable character, the `kubectl` command may fail.

The `kubectl` command does not print an error when it encounters an invalid filename.

Create the local directory:

```
mkdir -p configure-pod-container/configmap/
```

Now, download the sample configuration and create the ConfigMap:

```
# Download the sample files into `configure-pod-container/configmap/` directory
wget https://kubernetes.io/examples/configmap/game.properties -O configmap/game.properties
wget https://kubernetes.io/examples/configmap/ui.properties -O configmap/ui.properties

# Create the ConfigMap
kubectl create configmap game-config --from-file=configure-pod-container/configmap/
```

The above command packages each file, in this case, `game.properties` and `ui.properties` in the `configure-pod-container/configmap/` directory into the `game-config` ConfigMap. You can display details of the ConfigMap using the following command:

```
kubectl describe configmaps game-config
```

The output is similar to this:

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

Data
=====
game.properties:
-----
enemies=aliens
lives=3
enemies.cheat=true
enemies.cheat.level=noGoodRotten
secret.code.passphrase=UUDDLRLRBABAS
secret.code.allowed=true
secret.code.lives=30
ui.properties:
-----
color.good=purple
color.bad=yellow
allow.textmode=true
how.nice.to.look=fairlyNice
```

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

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

The output is similar to this:

```
apiVersion: v1
kind: ConfigMap
metadata:
  creationTimestamp: 2022-02-18T18:52:05Z
  name: game-config
  namespace: default
  resourceVersion: "516"
  uid: b4952dc3-d670-11e5-8cd0-68f728db1985
data:
  game.properties: |
    enemies=aliens
    Lives=3
    enemies.cheat=true
    enemies.cheat.Level=noGoodRotten
    secret.code.passphrase=UUDDLRLRBABAS
    secret.code.allowed=true
    secret.code.lives=30
  ui.properties: |
    color.good=purple
    color.bad=yellow
    allow.textmode=true
    how.nice.to.Look=fairlyNice
```

Create ConfigMaps from files

You can use `kubectl create configmap` to create a ConfigMap from an individual file, or from multiple files.

For example,

```
kubectl create configmap game-config-2 --from-file=configure-pod-conta
```

would produce the following ConfigMap:

```
kubectl describe configmaps game-config-2
```

where the output is similar to this:

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

Data
=====
game.properties:
-----
enemies=aliens
lives=3
enemies.cheat=true
enemies.cheat.level=noGoodRotten
secret.code.passphrase=UUDDLRLRBABAS
secret.code.allowed=true
secret.code.lives=30
```

You can pass in the `--from-file` argument multiple times to create a ConfigMap from multiple data sources.

```
kubectl create configmap game-config-2 --from-file=configure-pod-conta
```

You can display details of the `game-config-2` ConfigMap using the following command:

```
kubectl describe configmaps game-config-2
```

The output is similar to this:

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

Data
=====
game.properties:
-----
enemies=aliens
lives=3
enemies.cheat=true
enemies.cheat.level=noGoodRotten
secret.code.passphrase=UUDDLRLRBABAS
secret.code.allowed=true
secret.code.lives=30
ui.properties:
-----
color.good=purple
color.bad=yellow
allow.textmode=true
how.nice.to.look=fairlyNice
```

Use the option `--from-env-file` to create a ConfigMap from an env-file, for example:

```
# Env-files contain a list of environment variables.
# These syntax rules apply:
#   Each line in an env file has to be in VAR=VAL format.
#   Lines beginning with # (i.e. comments) are ignored.
#   Blank lines are ignored.
#   There is no special handling of quotation marks (i.e. they will be

# Download the sample files into `configure-pod-container/configmap/` directory
wget https://kubernetes.io/examples/configmap/game-env-file.properties
wget https://kubernetes.io/examples/configmap/ui-env-file.properties -O ui-env-file.properties

# The env-file `game-env-file.properties` looks like below
cat configure-pod-container/configmap/game-env-file.properties
enemies=aliens
lives=3
allowed="true"

# This comment and the empty line above it are ignored
```

```
kubectl create configmap game-config-env-file \
--from-env-file=configure-pod-container/configmap/game-env-file
```

would produce a ConfigMap. View the ConfigMap:

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

the output is similar to:

```
apiVersion: v1
kind: ConfigMap
metadata:
  creationTimestamp: 2019-12-27T18:36:28Z
  name: game-config-env-file
  namespace: default
  resourceVersion: "809965"
  uid: d9d1ca5b-eb34-11e7-887b-42010a8002b8
data:
  allowed: '"true"'
  enemies: aliens
  lives: "3"
```

Starting with Kubernetes v1.23, `kubectl` supports the `--from-env-file` argument to be specified multiple times to create a ConfigMap from multiple data sources.

```
kubectl create configmap config-multi-env-files \
  --from-env-file=configure-pod-container/configmap/game-env-file \
  --from-env-file=configure-pod-container/configmap/ui-env-file.
```

would produce the following ConfigMap:

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

where the output is similar to this:

```
apiVersion: v1
kind: ConfigMap
metadata:
  creationTimestamp: 2019-12-27T18:38:34Z
  name: config-multi-env-files
  namespace: default
  resourceVersion: "810136"
  uid: 252c4572-eb35-11e7-887b-42010a8002b8
data:
  allowed: '"true"'
  color: purple
  enemies: aliens
  how: fairlyNice
  lives: "3"
  textmode: "true"
```

Define the key to use when creating a ConfigMap from a file

You can define a key other than the file name to use in the `data` section of your ConfigMap when using the `--from-file` argument:

```
kubectl create configmap game-config-3 --from-file=<my-key-name>=<path
```

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=co
```

would produce the following ConfigMap:

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

where the output is similar to this:

```
apiVersion: v1
kind: ConfigMap
metadata:
  creationTimestamp: 2022-02-18T18:54:22Z
  name: game-config-3
  namespace: default
  resourceVersion: "530"
  uid: 05f8da22-d671-11e5-8cd0-68f728db1985
data:
  game-special-key: |
    enemies=aliens
    Lives=3
    enemies.cheat=true
    enemies.cheat.Level=noGoodRotten
    secret.code.passphrase=UUDDLRLRBABAS
    secret.code.allowed=true
    secret.code.Lives=30
```

Create ConfigMaps from literal values

You can use `kubectl create configmap` with the `--from-literal` argument to define a literal value from the command line:

```
kubectl create configmap special-config --from-literal=special.how=ver
```

You can pass in multiple key-value pairs. Each pair provided on the command line is represented as a separate entry in the `data` section of the ConfigMap.

```
kubectl get configmaps special-config -o yaml
```

The output is similar to this:

```
apiVersion: v1
kind: ConfigMap
metadata:
  creationTimestamp: 2022-02-18T19:14:38Z
  name: special-config
  namespace: default
  resourceVersion: "651"
  uid: dadce046-d673-11e5-8cd0-68f728db1985
data:
  special.how: very
  special.type: charm
```

Create a ConfigMap from generator

You can also create a ConfigMap from generators and then apply it to create the object in the cluster's API server. You should specify the generators in a `kustomization.yaml` file within a directory.

Generate ConfigMaps from files

For example, to generate a ConfigMap from files `configure-pod-container/configmap/game.properties`

```
# Create a kustomization.yaml file with ConfigMapGenerator
cat <<EOF >./kustomization.yaml
configMapGenerator:
- name: game-config-4
  options:
    labels:
      game-config: config-4
  files:
  - configure-pod-container/configmap/game.properties
EOF
```

Apply the kustomization directory to create the ConfigMap object:

```
kubectl apply -k .
```

```
configmap/game-config-4-m9dm2f92bt created
```

You can check that the ConfigMap was created like this:

```
kubectl get configmap
```

NAME	DATA	AGE
game-config-4-m9dm2f92bt	1	37s

and also:

```
kubectl describe configmaps/game-config-4-m9dm2f92bt
```

```
Name: game-config-4-m9dm2f92bt
Namespace: default
Labels: game-config=config-4
Annotations: kubectl.kubernetes.io/last-applied-configuration:
  {"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
Events: <none>"}}
```

Notice that the generated ConfigMap name has a suffix appended by hashing the contents. This ensures that a new ConfigMap is generated each time the content is modified.

Define the key to use when generating a ConfigMap from a file

You can define a key other than the file name to use in the ConfigMap generator. For example, to generate a ConfigMap from files `configure-pod-container/configmap/game.properties` with the key `game-special-key`

```
# Create a kustomization.yaml file with ConfigMapGenerator
cat <<EOF >./kustomization.yaml
configMapGenerator:
- name: game-config-5
  options:
    labels:
      game-config: config-5
  files:
    - game-special-key=configure-pod-container/configmap/game.properties
EOF
```

Apply the kustomization directory to create the ConfigMap object.

```
kubectl apply -k .
```

```
configmap/game-config-5-m67dt67794 created
```

Generate ConfigMaps from literals

This example shows you how to create a `ConfigMap` from two literal key/value pairs: `special.type=charm` and `special.how=very`, using Kustomize and `kubectl`. To achieve this, you can specify the `ConfigMap` generator. Create (or replace) `kustomization.yaml` so that it has the following contents:

```
---  
# kustomization.yaml contents for creating a ConfigMap from Literals  
configMapGenerator:  
- name: special-config-2  
literals:  
- special.how=very  
- special.type=charm
```

Apply the kustomization directory to create the ConfigMap object:

```
kubectl apply -k .
```

```
configmap/special-config-2-c92b5mmcf2 created
```

Interim cleanup

Before proceeding, clean up some of the ConfigMaps you made:

```
kubectl delete configmap special-config  
kubectl delete configmap env-config  
kubectl delete configmap -l 'game-config in (config-4,config-5)'
```

Now that you have learned to define ConfigMaps, you can move on to the next section, and learn how to use these objects with Pods.

Define container environment variables using ConfigMap data

Define a container environment variable with data from a single ConfigMap

1. Define an environment variable as a key-value pair in a ConfigMap:

```
kubectl create configmap special-config --from-literal=special.ho
```

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

[pods/pod-single-configmap-env-variable.yaml](#) 

```

apiVersion: v1
kind: Pod
metadata:
  name: dapi-test-pod
spec:
  containers:
    - name: test-container
      image: registry.k8s.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 access
              name: special-config
              # Specify the key associated with the value
              key: special.how
  restartPolicy: Never

```

Create the Pod:

```
kubectl create -f https://kubernetes.io/examples/pods/pod-single-
```

Now, the Pod's output includes environment variable

SPECIAL_LEVEL_KEY=very .

Define container environment variables with data from multiple ConfigMaps

As with the previous example, create the ConfigMaps first. Here is the manifest you will use:

[configmap/configmaps.yaml](#) 

```

apiVersion: v1
kind: ConfigMap
metadata:
  name: special-config
  namespace: default
data:
  special.how: very
---
apiVersion: v1
kind: ConfigMap
metadata:
  name: env-config
  namespace: default
data:
  log_level: INFO

```

- Create the ConfigMap:

```
kubectl create -f https://kubernetes.io/examples/configmap/config
```

- Define the environment variables in the Pod specification.

[pods/pod-multiple-configmap-env-variable.yaml](#)

```
apiVersion: v1
kind: Pod
metadata:
  name: dapi-test-pod
spec:
  containers:
    - name: test-container
      image: registry.k8s.io/busybox
      command: [ "/bin/sh", "-c", "env" ]
      env:
        - name: SPECIAL_LEVEL_KEY
          valueFrom:
            configMapKeyRef:
              name: special-config
              key: special.how
        - name: LOG_LEVEL
          valueFrom:
            configMapKeyRef:
              name: env-config
              key: log_level
  restartPolicy: Never
```

Create the Pod:

```
kubectl create -f https://kubernetes.io/examples/pods/pod-multipl
```

Now, the Pod's output includes environment variables

SPECIAL_LEVEL_KEY=very and LOG_LEVEL=INFO .

Once you're happy to move on, delete that Pod:

```
kubectl delete pod dapi-test-pod --now
```

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

- Create a ConfigMap containing multiple key-value pairs.

[configmap/configmap-multikeys.yaml](#)

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

Create the ConfigMap:

```
kubectl create -f https://kubernetes.io/examples/configmap/config
```

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

[pods/pod-configmap-envFrom.yaml](#) 

```
apiVersion: v1
kind: Pod
metadata:
  name: dapi-test-pod
spec:
  containers:
    - name: test-container
      image: registry.k8s.io/busybox
      command: [ "/bin/sh", "-c", "env" ]
      envFrom:
        - configMapRef:
            name: special-config
  restartPolicy: Never
```

Create the Pod:

```
kubectl create -f https://kubernetes.io/examples/pods/pod-config
```

Now, the Pod's output includes environment variables
`SPECIAL_LEVEL=very` and `SPECIAL_TYPE=charm`.

Once you're happy to move on, delete that Pod:

```
kubectl delete pod dapi-test-pod --now
```

Use ConfigMap-defined environment variables in Pod commands

You can use ConfigMap-defined environment variables in the `command` and `args` of a container using the `$(VAR_NAME)` Kubernetes substitution syntax.

For example, the following Pod manifest:

```
pods/pod-configmap-env-var-valueFrom.yaml □
```

```
apiVersion: v1
kind: Pod
metadata:
  name: dapi-test-pod
spec:
  containers:
    - name: test-container
      image: registry.k8s.io/busybox
      command: [ "/bin/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
```

Create that Pod, by running:

```
kubectl create -f https://kubernetes.io/examples/pods/pod-configmap-env-var-valueFrom.yaml
```

That pod produces the following output from the `test-container` container:

```
kubectl logs dapi-test-pod
```

```
very charm
```

Once you're happy to move on, delete that Pod:

```
kubectl delete pod dapi-test-pod --now
```

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`:

```
configmap/configmap-multikeys.yaml □
```

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

Create the ConfigMap:

```
kubectl create -f https://kubernetes.io/examples/configmap/configmap-m
```

Populate a Volume with data stored in a ConfigMap

Add the ConfigMap name under the `volumes` section of the Pod specification. This adds the ConfigMap data to the directory specified as `volumeMounts.mountPath` (in this case, `/etc/config`). The `command` section lists directory files with names that match the keys in ConfigMap.

[pods/pod-configmap-volume.yaml](#) 

```
apiVersion: v1
kind: Pod
metadata:
  name: dapi-test-pod
spec:
  containers:
    - name: test-container
      image: registry.k8s.io/busybox
      command: [ "/bin/sh", "-c", "ls /etc/config/" ]
      volumeMounts:
        - name: config-volume
          mountPath: /etc/config
  volumes:
    - name: config-volume
      configMap:
        # Provide the name of the ConfigMap containing the files you want to add to the container
        name: special-config
  restartPolicy: Never
```

Create the Pod:

```
kubectl create -f https://kubernetes.io/examples/pods/pod-configmap-vo
```

When the pod runs, the command `ls /etc/config/` produces the output below:

```
SPECIAL_LEVEL  
SPECIAL_TYPE
```

Text data is exposed as files using the UTF-8 character encoding. To use some other character encoding, use `binaryData` (see [ConfigMap object](#) for more details).

Note: If there are any files in the `/etc/config` directory of that container image, the volume mount will make those files from the image inaccessible.

Once you're happy to move on, delete that Pod:

```
kubectl delete pod dapi-test-pod --now
```

Add ConfigMap data to a specific path in the Volume

Use the `path` field to specify the desired file path for specific ConfigMap items. In this case, the `SPECIAL_LEVEL` item will be mounted in the `config-volume` volume at `/etc/config/keys`.

[pods/pod-configmap-volume-specific-key.yaml](#) 

```
apiVersion: v1
kind: Pod
metadata:
  name: dapi-test-pod
spec:
  containers:
    - name: test-container
      image: registry.k8s.io/busybox
      command: [ "/bin/sh", "-c", "cat /etc/config/keys" ]
      volumeMounts:
        - name: config-volume
          mountPath: /etc/config
  volumes:
    - name: config-volume
      configMap:
        name: special-config
        items:
          - key: SPECIAL_LEVEL
            path: keys
  restartPolicy: Never
```

Create the Pod:

```
kubectl create -f https://kubernetes.io/examples/pods/pod-configmap-vo
```

When the pod runs, the command `cat /etc/config/keys` produces the output below:

```
very
```

Caution: Like before, all previous files in the [/etc/config](#) directory will be deleted.

Delete that Pod:

```
kubectl delete pod dapi-test-pod --now
```

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](#) guide explains the syntax.

Optional references

A ConfigMap reference may be marked *optional*. If the ConfigMap is non-existent, the mounted volume will be empty. If the ConfigMap exists, but the referenced key is non-existent, the path will be absent beneath the mount point. See [Optional ConfigMaps](#) for more details.

Mounted ConfigMaps are updated automatically

When a mounted ConfigMap is updated, the projected content is eventually updated too. This applies in the case where an optionally referenced ConfigMap comes into existence after a pod has started.

Kubelet checks whether the mounted ConfigMap is fresh on every periodic sync. However, it uses its local TTL-based cache for getting the current value of the ConfigMap. As a result, the total delay from the moment when the ConfigMap is updated to the moment when new keys are projected to the pod can be as long as kubelet sync period (1 minute by default) + TTL of ConfigMaps cache (1 minute by default) in kubelet. You can trigger an immediate refresh by updating one of the pod's annotations.

Note: A container using a ConfigMap as a [subPath](#) volume will not receive ConfigMap updates.

Understanding ConfigMaps and Pods

The ConfigMap API resource stores configuration data as key-value pairs. The data can be consumed in pods or provide the configurations for system components such as controllers. ConfigMap is similar to [Secrets](#), but provides a means of working with strings that don't contain sensitive information. Users and system components alike can store configuration data in ConfigMap.

Note: ConfigMaps should reference properties files, not replace them. Think of the ConfigMap as representing something similar to the Linux [/etc](#) directory and its contents. For example, if you create a [Kubernetes Volume](#) from a ConfigMap, each data item in the ConfigMap is represented by an individual file in the volume.

The ConfigMap's `data` field contains the configuration data. As shown in the example below, this can be simple (like individual properties defined using `--from-literal`) or complex (like configuration files or JSON blobs

defined using `--from-file`).

```
apiVersion: v1
kind: ConfigMap
metadata:
  creationTimestamp: 2016-02-18T19:14:38Z
  name: example-config
  namespace: default
data:
  # example of a simple property defined using --from-literal
  example.property.1: hello
  example.property.2: world
  # example of a complex property defined using --from-file
  example.property.file: |-  

    property.1=value-1
    property.2=value-2
    property.3=value-3
```

When `kubectl` creates a ConfigMap from inputs that are not ASCII or UTF-8, the tool puts these into the `binaryData` field of the ConfigMap, and not in `data`. Both text and binary data sources can be combined in one ConfigMap.

If you want to view the `binaryData` keys (and their values) in a ConfigMap, you can run `kubectl get configmap -o jsonpath='{.binaryData}' <name>`.

Pods can load data from a ConfigMap that uses either `data` or `binaryData`.

Optional ConfigMaps

You can mark a reference to a ConfigMap as *optional* in a Pod specification. If the ConfigMap doesn't exist, the configuration for which it provides data in the Pod (for example: environment variable, mounted volume) will be empty. If the ConfigMap exists, but the referenced key is non-existent the data is also empty.

For example, the following Pod specification marks an environment variable from a ConfigMap as optional:

```
apiVersion: v1
kind: Pod
metadata:
  name: dapi-test-pod
spec:
  containers:
    - name: test-container
      image: gcr.io/google_containers/busybox
      command: ["/bin/sh", "-c", "env"]
      env:
        - name: SPECIAL_LEVEL_KEY
          valueFrom:
            configMapKeyRef:
              name: a-config
              key: akey
              optional: true # mark the variable as optional
  restartPolicy: Never
```

If you run this pod, and there is no ConfigMap named `a-config`, the output is empty. If you run this pod, and there is a ConfigMap named `a-config` but that ConfigMap doesn't have a key named `akey`, the output is

also empty. If you do set a value for `key` in the `a-config` ConfigMap, this pod prints that value and then terminates.

You can also mark the volumes and files provided by a ConfigMap as optional. Kubernetes always creates the mount paths for the volume, even if the referenced ConfigMap or key doesn't exist. For example, the following Pod specification marks a volume that references a ConfigMap as optional:

```
apiVersion: v1
kind: Pod
metadata:
  name: dapi-test-pod
spec:
  containers:
    - name: test-container
      image: gcr.io/google_containers/busybox
      command: ["/bin/sh", "-c", "ls /etc/config"]
      volumeMounts:
        - name: config-volume
          mountPath: /etc/config
  volumes:
    - name: config-volume
      configMap:
        name: no-config
        optional: true # mark the source ConfigMap as optional
  restartPolicy: Never
```

Restrictions

- You must create the `ConfigMap` object before you reference it in a Pod specification. Alternatively, mark the ConfigMap reference as `optional` in the Pod spec (see [Optional ConfigMaps](#)). If you reference a ConfigMap that doesn't exist and you don't mark the reference as `optional`, the Pod won't start. Similarly, references to keys that don't exist in the ConfigMap will also prevent the Pod from starting, unless you mark the key references as `optional`.
- If you use `envFrom` to define environment variables from ConfigMaps, keys that are considered invalid will be skipped. The pod will be allowed to start, but the invalid names will be recorded in the event log (`InvalidVariableNames`). The log message lists each skipped key. For example:

```
kubectl get events
```

The output is similar to this:

LASTSEEN	FIRSTSEEN	COUNT	NAME	KIND	SUBOBJECT	TYPE
0s	0s	1	dapi-test-pod	Pod		Warning

- ConfigMaps reside in a specific Namespace. Pods can only refer to ConfigMaps that are in the same namespace as the Pod.
- You can't use ConfigMaps for static pods, because the kubelet does not support this.

Cleaning up

Delete the ConfigMaps and Pods that you made:

```
kubectl delete configmaps/game-config configmaps/game-config-2 configmaps/game-config-env-file  
kubectl delete pod dapi-test-pod --now  
  
# You might already have removed the next set  
kubectl delete configmaps/special-config configmaps/env-config  
kubectl delete configmap -l 'game-config in (config-4,config-5)'
```

If you created a directory `configure-pod-container` and no longer need it, you should remove that too, or move it into the trash can / deleted files location.

What's next

- Follow a real world example of [Configuring Redis using a ConfigMap](#).

3.21 - Share Process Namespace between Containers in a Pod

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 the same 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

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

- [KillerCoda](#)
- [Play with Kubernetes](#)

Configure a Pod

Process namespace sharing is enabled using the `shareProcessNamespace` field of `.spec` for a Pod. For example:

```
pods/share-process-namespace.yaml 
```

```
apiVersion: v1
kind: Pod
metadata:
  name: nginx
spec:
  shareProcessNamespace: true
  containers:
    - name: nginx
      image: nginx
    - name: shell
      image: busybox:1.28
      command: ["sleep", "3600"]
  securityContext:
    capabilities:
      add:
        - SYS_PTRACE
  stdin: true
  tty: true
```

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

```
kubectl apply -f https://k8s.io/examples/pods/share-process-names
```

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. In the container shell:

```
# run this inside the "shell" container
ps ax
```

The output is similar to this:

```
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.

```
# run this inside the "shell" container
kill -HUP 8  # change "8" to match the PID of the nginx Leader process
ps ax
```

The output is similar to this:

```
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 the file system of another container using the /proc/\$pid/root link.

```
# run this inside the "shell" container
# change "8" to the PID of the Nginx process, if necessary
head /proc/8/root/etc/nginx/nginx.conf
```

The output is similar to this:

```
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 containers may expect to be isolated from others, though, so it's important to understand the differences:

1. **The container process no longer has PID 1.** Some containers 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.

3.22 - Use a User Namespace With a Pod

FEATURE STATE: [Kubernetes v1.25 \[alpha\]](#)

This page shows how to configure a user namespace for pods. This allows you to isolate the user running inside the container from the one in the host.

A process running as root in a container can run as a different (non-root) user in the host; in other words, the process has full privileges for operations inside the user namespace, but is unprivileged for operations outside the namespace.

You can use this feature to reduce the damage a compromised container can do to the host or other pods in the same node. There are [several security vulnerabilities](#) rated either **HIGH** or **CRITICAL** that were not exploitable when user namespaces is active. It is expected user namespace will mitigate some future vulnerabilities too.

Without using a user namespace a container running as root, in the case of a container breakout, has root privileges on the node. And if some capability were granted to the container, the capabilities are valid on the host too. None of this is true when user namespaces are used.

Before you begin

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

- [Killercoda](#)
- [Play with Kubernetes](#)

Your Kubernetes server must be at or later than version v1.25. To check the version, enter [kubectl version](#).

ⓘ This item links to a third party project or product that is not part of Kubernetes itself. [More information](#)

- The node OS needs to be Linux
- You need to exec commands in the host
- You need to be able to exec into pods
- You need to enable the `UserNamespacesSupport` [feature gate](#)

Note: The feature gate to enable user namespaces was previously named `UserNamespacesStatelessPodsSupport`, when only stateless pods were supported. Only Kubernetes v1.25 through to v1.27 recognise `UserNamespacesStatelessPodsSupport`.

The cluster that you're using **must** include at least one node that meets the [requirements](#) for using user namespaces with Pods.

If you have a mixture of nodes and only some of the nodes provide user namespace support for Pods, you also need to ensure that the user namespace Pods are [scheduled](#) to suitable nodes.

Please note that **if your container runtime doesn't support user namespaces, the `hostUsers` field in the pod spec will be silently ignored and the pod will be created without user namespaces.**

Run a Pod that uses a user namespace

A user namespace for a pod is enabled setting the `hostUsers` field of `.spec` to `false`. For example:

```
apiVersion: v1
```

```
kind: Pod
```

```
metadata:
```

```
  name: userns
```

```
spec:
```

```
  hostUsers: false
```

```
  containers:
```

```
  - name: shell
```

```
    command: ["sleep", "infinity"]
```

```
    image: debian
```

1. Create the pod on your cluster:

```
kubectl apply -f https://k8s.io/examples/pods/user-namespaces-sta
```

2. Attach to the container and run `readlink /proc/self/ns/user`:

```
kubectl attach -it userns bash
```

And run the command. The output is similar to this:

```
readlink /proc/self/ns/user
user:[4026531837]
cat /proc/self/uid_map
0          0 4294967295
```

Then, open a shell in the host and run the same command.

The output must be different. This means the host and the pod are using a different user namespace. When user namespaces are not enabled, the host and the pod use the same user namespace.

If you are running the kubelet inside a user namespace, you need to compare the output from running the command in the pod to the output of running in the host:

```
readlink /proc/$pid/ns/user
user:[4026534732]
```

replacing `$pid` with the kubelet PID.

3.23 - Create static Pods

Static Pods are managed directly by the kubelet daemon on a specific node, without the API server observing them. Unlike Pods that are managed by the control plane (for example, a [Deployment](#)); instead, the kubelet watches each static Pod (and restarts it if it fails).

Static Pods are always bound to one Kubelet on a specific node.

The kubelet automatically tries to create a [mirror Pod](#) on the Kubernetes API server for each static Pod. This means that the Pods running on a node are visible on the API server, but cannot be controlled from there. The Pod names will be suffixed with the node hostname with a leading hyphen.

Note: If you are running clustered Kubernetes and are using static Pods to run a Pod on every node, you should probably be using a [DaemonSet](#) instead.

Note: The [spec](#) of a static Pod cannot refer to other API objects (e.g., [ServiceAccount](#), [ConfigMap](#), [Secret](#), etc).

Note: Static pods do not support [ephemeral containers](#).

Before you begin

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

- [Killercoda](#)
- [Play with Kubernetes](#)

To check the version, enter `kubectl version`.

This page assumes you're using [CRI-O](#) to run Pods, and that your nodes are running the Fedora operating system. Instructions for other distributions or Kubernetes installations may vary.

Create a static pod

You can configure a static Pod with either a [file system hosted configuration file](#) or a [web hosted configuration file](#).

Filesystem-hosted static Pod manifest

Manifests are standard Pod definitions in JSON or YAML format in a specific directory. Use the `staticPodPath: <the directory>` field in the [kubelet configuration file](#), which periodically scans the directory and creates/deletes static Pods as YAML/JSON files appear/disappear there. Note that the kubelet will ignore files starting with dots when scanning the specified directory.

For example, this is how to start a simple web server as a static Pod:

1. Choose a node where you want to run the static Pod. In this example, it's `my-node1`.

```
ssh my-node1
```

2. Choose a directory, say `/etc/kubernetes/manifests` and place a web server Pod definition there, for example `/etc/kubernetes/manifests/static-web.yaml`:

```
# Run this command on the node where kubelet is running
mkdir -p /etc/kubernetes/manifests/
cat <<EOF >/etc/kubernetes/manifests/static-web.yaml
apiVersion: v1
kind: Pod
metadata:
  name: static-web
  labels:
    role: myrole
spec:
  containers:
    - name: web
      image: nginx
      ports:
        - name: web
          containerPort: 80
          protocol: TCP
EOF
```

3. Configure your kubelet on the node to use this directory by running it with `--pod-manifest-path=/etc/kubernetes/manifests/` argument. On Fedora, edit `/etc/kubernetes/kubelet` to include this line:

```
KUBELET_ARGS="--cluster-dns=10.254.0.10 --cluster-domain=kube.loc:
```

or add the `staticPodPath: <the directory>` field in the [kubelet configuration file](#).

4. Restart the kubelet. On Fedora, you would run:

```
# Run this command on the node where the kubelet is running
systemctl restart kubelet
```

Web-hosted static pod manifest

Kubelet periodically downloads a file specified by `--manifest-url=<URL>` argument and interprets it as a JSON/YAML file that contains Pod definitions. Similar to how [filesystem-hosted manifests](#) work, the kubelet refetches the manifest on a schedule. If there are changes to the list of static Pods, the kubelet applies them.

To use this approach:

1. Create a YAML file and store it on a web server so that you can pass the URL of that file to the kubelet.

```
apiVersion: v1
kind: Pod
metadata:
  name: static-web
  labels:
    role: myrole
spec:
  containers:
    - name: web
      image: nginx
      ports:
        - name: web
          containerPort: 80
          protocol: TCP
```

2. Configure the kubelet on your selected node to use this web manifest by running it with `--manifest-url=<manifest-url>`. On Fedora, edit `/etc/kubernetes/kubelet` to include this line:

```
KUBELET_ARGS="--cluster-dns=10.254.0.10 --cluster-domain=kube.loc
```

3. Restart the kubelet. On Fedora, you would run:

```
# Run this command on the node where the kubelet is running
systemctl restart kubelet
```

Observe static pod behavior

When the kubelet starts, it automatically starts all defined static Pods. As you have defined a static Pod and restarted the kubelet, the new static Pod should already be running.

You can view running containers (including static Pods) by running (on the node):

```
# Run this command on the node where the kubelet is running
crictl ps
```

The output might be something like:

CONTAINER	IMAGE	CREATED
129fd7d382018	docker.io/library/nginx@sha256:...	11 minutes ago

Note: `crictl` outputs the image URI and SHA-256 checksum. `NAME` will look more like:
`docker.io/library/nginx@sha256:0d17b565c37bcd895e9d92315a05c1c3c9a29f762b011a10c54a66cd53c9b31.`

You can see the mirror Pod on the API server:

```
kubectl get pods
```

NAME	READY	STATUS	RESTARTS	AGE
static-web-my-node1	1/1	Running	0	2m

Note: Make sure the kubelet has permission to create the mirror Pod in the API server. If not, the creation request is rejected by the API server.

Labels from the static Pod are propagated into the mirror Pod. You can use those labels as normal via selectors, etc.

If you try to use `kubectl` to delete the mirror Pod from the API server, the kubelet *doesn't* remove the static Pod:

```
kubectl delete pod static-web-my-node1
```

```
pod "static-web-my-node1" deleted
```

You can see that the Pod is still running:

```
kubectl get pods
```

NAME	READY	STATUS	RESTARTS	AGE
static-web-my-node1	1/1	Running	0	4s

Back on your node where the kubelet is running, you can try to stop the container manually. You'll see that, after a time, the kubelet will notice and will restart the Pod automatically:

```
# Run these commands on the node where the kubelet is running
crictl stop 129fd7d382018 # replace with the ID of your container
sleep 20
crictl ps
```

CONTAINER	IMAGE	CREATED
89db4553e1eeb	docker.io/library/nginx@sha256:...	19 seconds ago

Once you identify the right container, you can get the logs for that container with `crictl`:

```
# Run these commands on the node where the container is running
crictl logs <container_id>
```

```
10.240.0.48 - - [16/Nov/2022:12:45:49 +0000] "GET / HTTP/1.1" 200 612
10.240.0.48 - - [16/Nov/2022:12:45:50 +0000] "GET / HTTP/1.1" 200 612
10.240.0.48 - - [16/Nov/2022:12:45:51 +0000] "GET / HTTP/1.1" 200 612
```

To find more about how to debug using `cricctl`, please visit [Debugging Kubernetes nodes with crictl](#).

Dynamic addition and removal of static pods

The running kubelet periodically scans the configured directory (`/etc/kubernetes/manifests` in our example) for changes and adds/removes Pods as files appear/disappear in this directory.

```
# This assumes you are using filesystem-hosted static Pod configuration
# Run these commands on the node where the container is running
#
mv /etc/kubernetes/manifests/static-web.yaml /tmp
sleep 20
cricctl ps
# You see that no nginx container is running
mv /tmp/static-web.yaml /etc/kubernetes/manifests/
sleep 20
cricctl ps
```

CONTAINER	IMAGE	CREATED
f427638871c35	docker.io/library/nginx@sha256:...	19 seconds ago

What's next

- [Generate static Pod manifests for control plane components](#)
- [Generate static Pod manifest for local etcd](#)
- [Debugging Kubernetes nodes with crictl](#)
- [Learn more about crictl](#).
- [Map docker CLI commands to crictl](#).
- [Set up etcd instances as static pods managed by a kubelet](#)

3.24 - 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

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

- [Killercoda](#)
- [Play with Kubernetes](#)

To check the version, enter `kubectl version`.

Install Kompose

We have multiple ways to install Kompose. Our preferred method is downloading the binary from the latest GitHub release.

[GitHub download](#) [Build from source](#) [CentOS package](#)
[Fedora package](#) [Homebrew \(macOS\)](#)

Kompose is released via GitHub on a three-week cycle, you can see all current releases on the [GitHub release page](#).

```
# Linux
curl -L https://github.com/kubernetes/kompose/releases/download/v1.26.0/kompose-linu

# macOS
curl -L https://github.com/kubernetes/kompose/releases/download/v1.26.0/kompose-darw

# Windows
curl -L https://github.com/kubernetes/kompose/releases/download/v1.26.0/kompose-wind

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

Alternatively, you can download the [tarball](#).

Use Kompose

In 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: registry.k8s.io/redis:e2e
    ports:
      - "6379"

  redis-slave:
    image: gcr.io/google_samples/gb-redisslave:v3
    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. To convert the `docker-compose.yml` file to files that you can use with `kubectl`, run `kompose convert` and then `kubectl apply -f <output file>`.

```
kompose convert
```

The output is similar to:

```
INFO Kubernetes file "frontend-tcp-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 apply -f frontend-tcp-service.yaml,redis-master-service.
```

The output is similar to:

```
service/frontend-tcp created
service/redis-master created
service/redis-slave created
deployment.apps/frontend created
deployment.apps/redis-master created
deployment.apps/redis-slave created
```

Your deployments are running in Kubernetes.

3. 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-tcp
Namespace: default
Labels: io.kompose.service=frontend-tcp
Annotations: kompose.cmd: kompose convert
kompose.service.type: LoadBalancer
kompose.version: 1.26.0 (40646f47)
Selector: io.kompose.service=frontend
Type: LoadBalancer
IP Family Policy: SingleStack
IP Families: IPv4
IP: 10.43.67.174
IPs: 10.43.67.174
Port: 80 80/TCP
TargetPort: 80/TCP
NodePort: 80 31254/TCP
Endpoints: 10.42.0.25:80
Session Affinity: None
External Traffic Policy: Cluster
Events:
  Type Reason Age From Message
  ---- ---- - - -
  Normal EnsuringLoadBalancer 62s service-controller Ensuring
  Normal AppliedDaemonSet 62s service-controller Applied
```

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

```
curl http://192.0.2.89
```

4. Clean-up.

After you are finished testing out the example application deployment, simply run the following command in your shell to delete the resources used.

```
kubectl delete -f frontend-tcp-service.yaml,redis-master-service.
```

User Guide

- CLI
 - [kompose convert](#)
- Documentation
 - [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 convert example

```
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
db-svc.yaml        docker-voting.yml       redis-svc.yaml    result
          ◀          ▶
```

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      re
frontend-deployment.yaml  mongodb-claim0-persistentvolumeclaim.yaml  re
frontend-service.yaml     mongodb-deployment.yaml      re
redis-master-deployment.yaml
```

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

OpenShift `kompose convert` example

```
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 conv
```

```
WARN [foo] Service cannot be created because of missing port.
INFO OpenShift Buildconfig using git@github.com:rtnpro/kompose.git::master
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>.

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 DaemonSet objects.

If you want to generate a Chart to be used with [Helm](#) run:

```
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
<code>kompose.service.type</code>	nodeport / clusterip / loadbalancer
<code>kompose.service.expose</code>	true / hostname

Note: The `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>	<code>object created</code>	<code>Pod restartPolicy</code>
""	controller object	Always
always	controller object	Always
on-failure	Pod	OnFailure
no	Pod	Never

Note: The controller object could be `deployment` or `replicationcontroller`.

For example, the `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 Configurations

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 (for example, `web_service`), then it will be replaced by `-` and the service name will be renamed accordingly (for example, `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.

3.25 - Enforce Pod Security Standards by Configuring the Built-in Admission Controller

Kubernetes provides a built-in [admission controller](#) to enforce the [Pod Security Standards](#). You can configure this admission controller to set cluster-wide defaults and [exemptions](#).

Before you begin

Following an alpha release in Kubernetes v1.22, Pod Security Admission became available by default in Kubernetes v1.23, as a beta. From version 1.25 onwards, Pod Security Admission is generally available.

To check the version, enter `kubectl version`.

If you are not running Kubernetes 1.28, you can switch to viewing this page in the documentation for the Kubernetes version that you are running.

Configure the Admission Controller

Note: `pod-security.admission.config.k8s.io/v1` configuration requires v1.25+. For v1.23 and v1.24, use [v1beta1](#). For v1.22, use [v1alpha1](#).

```
apiVersion: apiserver.config.k8s.io/v1
kind: AdmissionConfiguration
plugins:
- name: PodSecurity
  configuration:
    apiVersion: pod-security.admission.config.k8s.io/v1 # see compatibility notes
    kind: PodSecurityConfiguration
    # Defaults applied when a mode Label is not set.
    #
    # Level Label values must be one of:
    # - "privileged" (default)
    # - "baseline"
    # - "restricted"
    #
    # Version Label values must be one of:
    # - "Latest" (default)
    # - specific version like "v1.28"
  defaults:
    enforce: "privileged"
    enforce-version: "latest"
    audit: "privileged"
    audit-version: "latest"
    warn: "privileged"
    warn-version: "latest"
  exemptions:
    # Array of authenticated usernames to exempt.
    usernames: []
    # Array of runtime class names to exempt.
    runtimeClasses: []
    # Array of namespaces to exempt.
    namespaces: []
```

Note: The above manifest needs to be specified via the `--admission-control-config-file` to kube-apiserver.

3.26 - Enforce Pod Security Standards with Namespace Labels

Namespaces can be labeled to enforce the [Pod Security Standards](#). The three policies [privileged](#), [baseline](#) and [restricted](#) broadly cover the security spectrum and are implemented by the [Pod Security](#) admission controller.

Before you begin

Pod Security Admission was available by default in Kubernetes v1.23, as a beta. From version 1.25 onwards, Pod Security Admission is generally available.

To check the version, enter `kubectl version`.

Requiring the [baseline](#) Pod Security Standard with namespace labels

This manifest defines a Namespace `my-baseline-namespace` that:

- *Blocks* any pods that don't satisfy the `baseline` policy requirements.
- Generates a user-facing warning and adds an audit annotation to any created pod that does not meet the `restricted` policy requirements.
- Pins the versions of the `baseline` and `restricted` policies to v1.28.

```
apiVersion: v1
kind: Namespace
metadata:
  name: my-baseline-namespace
labels:
  pod-security.kubernetes.io/enforce: baseline
  pod-security.kubernetes.io/enforce-version: v1.28

  # We are setting these to our _desired_ `enforce` level.
  pod-security.kubernetes.io/audit: restricted
  pod-security.kubernetes.io/audit-version: v1.28
  pod-security.kubernetes.io/warn: restricted
  pod-security.kubernetes.io/warn-version: v1.28
```

Add labels to existing namespaces with [kubectl label](#)

Note: When an `enforce` policy (or version) label is added or changed, the admission plugin will test each pod in the namespace against the new policy. Violations are returned to the user as warnings.

It is helpful to apply the `--dry-run` flag when initially evaluating security profile changes for namespaces. The Pod Security Standard checks will still be run in *dry run* mode, giving you information about how the new policy would treat existing pods, without actually updating a policy.

```
kubectl label --dry-run=server --overwrite ns --all \
  pod-security.kubernetes.io/enforce=baseline
```

Applying to all namespaces

If you're just getting started with the Pod Security Standards, a suitable first step would be to configure all namespaces with audit annotations for a stricter level such as `baseline`:

```
kubectl label --overwrite ns --all \
  pod-security.kubernetes.io/audit=baseline \
  pod-security.kubernetes.io/warn=baseline
```

Note that this is not setting an enforce level, so that namespaces that haven't been explicitly evaluated can be distinguished. You can list namespaces without an explicitly set enforce level using this command:

```
kubectl get namespaces --selector='!pod-security.kubernetes.io/enforce'
```

Applying to a single namespace

You can update a specific namespace as well. This command adds the `enforce=restricted` policy to `my-existing-namespace`, pinning the restricted policy version to v1.28.

```
kubectl label --overwrite ns my-existing-namespace \
  pod-security.kubernetes.io/enforce=restricted \
  pod-security.kubernetes.io/enforce-version=v1.28
```

3.27 - Migrate from PodSecurityPolicy to the Built-In PodSecurity Admission Controller

This page describes the process of migrating from PodSecurityPolicies to the built-in PodSecurity admission controller. This can be done effectively using a combination of dry-run and `audit` and `warn` modes, although this becomes harder if mutating PSPs are used.

Before you begin

Your Kubernetes server must be at or later than version v1.22. To check the version, enter `kubectl version`.

If you are currently running a version of Kubernetes other than 1.28, you may want to switch to viewing this page in the documentation for the version of Kubernetes that you are actually running.

This page assumes you are already familiar with the basic [Pod Security Admission](#) concepts.

Overall approach

There are multiple strategies you can take for migrating from PodSecurityPolicy to Pod Security Admission. The following steps are one possible migration path, with a goal of minimizing both the risks of a production outage and of a security gap.

0. Decide whether Pod Security Admission is the right fit for your use case.
1. Review namespace permissions
2. Simplify & standardize PodSecurityPolicies
3. Update namespaces
 1. Identify an appropriate Pod Security level
 2. Verify the Pod Security level
 3. Enforce the Pod Security level
 4. Bypass PodSecurityPolicy
4. Review namespace creation processes
5. Disable PodSecurityPolicy

0. Decide whether Pod Security Admission is right for you

Pod Security Admission was designed to meet the most common security needs out of the box, and to provide a standard set of security levels across clusters. However, it is less flexible than PodSecurityPolicy. Notably, the following features are supported by PodSecurityPolicy but not Pod Security Admission:

- **Setting default security constraints** - Pod Security Admission is a non-mutating admission controller, meaning it won't modify pods before validating them. If you were relying on this aspect of PSP, you will need to either modify your workloads to meet the Pod Security constraints, or use a [Mutating Admission Webhook](#) to make those

changes. See [Simplify & Standardize PodSecurityPolicies](#) below for more detail.

- **Fine-grained control over policy definition** - Pod Security Admission only supports [3 standard levels](#). If you require more control over specific constraints, then you will need to use a [Validating Admission Webhook](#) to enforce those policies.
- **Sub-namespace policy granularity** - PodSecurityPolicy lets you bind different policies to different Service Accounts or users, even within a single namespace. This approach has many pitfalls and is not recommended, but if you require this feature anyway you will need to use a 3rd party webhook instead. The exception to this is if you only need to completely exempt specific users or [RuntimeClasses](#), in which case Pod Security Admission does expose some [static configuration for exemptions](#).

Even if Pod Security Admission does not meet all of your needs it was designed to be *complementary* to other policy enforcement mechanisms, and can provide a useful fallback running alongside other admission webhooks.

1. Review namespace permissions

Pod Security Admission is controlled by [labels on namespaces](#). This means that anyone who can update (or patch or create) a namespace can also modify the Pod Security level for that namespace, which could be used to bypass a more restrictive policy. Before proceeding, ensure that only trusted, privileged users have these namespace permissions. It is not recommended to grant these powerful permissions to users that shouldn't have elevated permissions, but if you must you will need to use an [admission webhook](#) to place additional restrictions on setting Pod Security labels on Namespace objects.

2. Simplify & standardize PodSecurityPolicies

In this section, you will reduce mutating PodSecurityPolicies and remove options that are outside the scope of the Pod Security Standards. You should make the changes recommended here to an offline copy of the original PodSecurityPolicy being modified. The cloned PSP should have a different name that is alphabetically before the original (for example, prepend a `0` to it). Do not create the new policies in Kubernetes yet - that will be covered in the [Rollout the updated policies](#) section below.

2.a. Eliminate purely mutating fields

If a PodSecurityPolicy is mutating pods, then you could end up with pods that don't meet the Pod Security level requirements when you finally turn PodSecurityPolicy off. In order to avoid this, you should eliminate all PSP mutation prior to switching over. Unfortunately PSP does not cleanly separate mutating & validating fields, so this is not a straightforward migration.

You can start by eliminating the fields that are purely mutating, and don't have any bearing on the validating policy. These fields (also listed in the [Mapping PodSecurityPolicies to Pod Security Standards](#) reference) are:

- `.spec.defaultAllowPrivilegeEscalation`
- `.spec.runtimeClass.defaultRuntimeClassName`
- `.metadata.annotations['seccomp.security.alpha.kubernetes.io/defaultProfileName']`

- `.metadata.annotations['apparmor.security.beta.kubernetes.io/defaultProfileName']`
- `.spec.defaultAddCapabilities` - Although technically a mutating & validating field, these should be merged into `.spec.allowedCapabilities` which performs the same validation without mutation.

Caution: Removing these could result in workloads missing required configuration, and cause problems. See [Rollout the updated policies](#) below for advice on how to roll these changes out safely.

2.b. Eliminate options not covered by the Pod Security Standards

There are several fields in PodSecurityPolicy that are not covered by the Pod Security Standards. If you must enforce these options, you will need to supplement Pod Security Admission with an [admission webhook](#), which is outside the scope of this guide.

First, you can remove the purely validating fields that the Pod Security Standards do not cover. These fields (also listed in the [Mapping PodSecurityPolicies to Pod Security Standards](#) reference with "no opinion") are:

- `.spec.allowedHostPaths`
- `.spec.allowedFlexVolumes`
- `.spec.allowedCSIDrivers`
- `.spec.forbiddenSysctls`
- `.spec.runtimeClass`

You can also remove the following fields, that are related to POSIX / UNIX group controls.

Caution: If any of these use the `MustRunAs` strategy they may be mutating! Removing these could result in workloads not setting the required groups, and cause problems. See [Rollout the updated policies](#) below for advice on how to roll these changes out safely.

- `.spec.runAsGroup`
- `.spec.supplementalGroups`
- `.spec.fsGroup`

The remaining mutating fields are required to properly support the Pod Security Standards, and will need to be handled on a case-by-case basis later:

- `.spec.requiredDropCapabilities` - Required to drop `ALL` for the Restricted profile.
- `.spec.seLinux` - (Only mutating with the `MustRunAs` rule) required to enforce the SELinux requirements of the Baseline & Restricted profiles.
- `.spec.runAsUser` - (Non-mutating with the `RunAsAny` rule) required to enforce `RunAsNonRoot` for the Restricted profile.
- `.spec.allowPrivilegeEscalation` - (Only mutating if set to `false`) required for the Restricted profile.

2.c. Rollout the updated PSPs

Next, you can rollout the updated policies to your cluster. You should proceed with caution, as removing the mutating options may result in workloads missing required configuration.

For each updated PodSecurityPolicy:

1. Identify pods running under the original PSP. This can be done using the `kubernetes.io/psp` annotation. For example, using kubectl:

```
PSP_NAME="original" # Set the name of the PSP you're checking for
kubectl get pods --all-namespaces -o jsonpath="{range .items[?(@.Annotations['kubernetes.io/psp'] == 'original')]}{.metadata.name}{end}"
```

2. Compare these running pods against the original pod spec to determine whether PodSecurityPolicy has modified the pod. For pods created by a [workload resource](#) you can compare the pod with the PodTemplate in the controller resource. If any changes are identified, the original Pod or PodTemplate should be updated with the desired configuration. The fields to review are:

- `.metadata.annotations['container.apparmor.security.beta.kubernetes.io/*']` (replace * with each container name)
- `.spec.runtimeClassName`
- `.spec.securityContext.fsGroup`
- `.spec.securityContext.seccompProfile`
- `.spec.securityContext.seLinuxOptions`
- `.spec.securityContext.supplementalGroups`
- On containers, under `.spec.containers[*]` and `.spec.initContainers[*]`:
 - `.securityContext.allowPrivilegeEscalation`
 - `.securityContext.capabilities.add`
 - `.securityContext.capabilities.drop`
 - `.securityContext.readOnlyRootFilesystem`
 - `.securityContext.runAsGroup`
 - `.securityContext.runAsNonRoot`
 - `.securityContext.runAsUser`
 - `.securityContext.seccompProfile`
 - `.securityContext.seLinuxOptions`

3. Create the new PodSecurityPolicies. If any Roles or ClusterRoles are granting `use` on all PSPs this could cause the new PSPs to be used instead of their mutating counter-parts.
4. Update your authorization to grant access to the new PSPs. In RBAC this means updating any Roles or ClusterRoles that grant the `use` permission on the original PSP to also grant it to the updated PSP.
5. Verify: after some soak time, rerun the command from step 1 to see if any pods are still using the original PSPs. Note that pods need to be recreated after the new policies have been rolled out before they can be fully verified.
6. (optional) Once you have verified that the original PSPs are no longer in use, you can delete them.

3. Update Namespaces

The following steps will need to be performed on every namespace in the cluster. Commands referenced in these steps use the `$NAMESPACE` variable to refer to the namespace being updated.

3.a. Identify an appropriate Pod Security level

Start reviewing the [Pod Security Standards](#) and familiarizing yourself with the 3 different levels.

There are several ways to choose a Pod Security level for your namespace:

1. **By security requirements for the namespace** - If you are familiar with the expected access level for the namespace, you can choose an appropriate level based on those requirements, similar to how one might approach this on a new cluster.
2. **By existing PodSecurityPolicies** - Using the [Mapping PodSecurityPolicies to Pod Security Standards](#) reference you can map each PSP to a Pod Security Standard level. If your PSPs aren't based on the Pod Security Standards, you may need to decide between choosing a level that is at least as permissive as the PSP, and a level that is at least as restrictive. You can see which PSPs are in use for pods in a given namespace with this command:

```
kubectl get pods -n $NAMESPACE -o jsonpath="{.items[*].metadata.annotations.pod-security.level}"
```

3. **By existing pods** - Using the strategies under [Verify the Pod Security level](#), you can test out both the Baseline and Restricted levels to see whether they are sufficiently permissive for existing workloads, and chose the least-privileged valid level.

Caution: Options 2 & 3 above are based on *existing* pods, and may miss workloads that aren't currently running, such as CronJobs, scale-to-zero workloads, or other workloads that haven't rolled out.

3.b. Verify the Pod Security level

Once you have selected a Pod Security level for the namespace (or if you're trying several), it's a good idea to test it out first (you can skip this step if using the Privileged level). Pod Security includes several tools to help test and safely roll out profiles.

First, you can dry-run the policy, which will evaluate pods currently running in the namespace against the applied policy, without making the new policy take effect:

```
# $LEVEL is the level to dry-run, either "baseline" or "restricted".
kubectl label --dry-run=server --overwrite ns $NAMESPACE pod-security.$LEVEL
```

This command will return a warning for any *existing* pods that are not valid under the proposed level.

The second option is better for catching workloads that are not currently running: audit mode. When running under audit-mode (as opposed to enforcing), pods that violate the policy level are recorded in the audit logs, which can be reviewed later after some soak time, but are not forbidden. Warning mode works similarly, but returns the warning to the user immediately. You can set the audit level on a namespace with this command:

```
kubectl label --overwrite ns $NAMESPACE pod-security.kubernetes.io/aud
```

If either of these approaches yield unexpected violations, you will need to either update the violating workloads to meet the policy requirements, or relax the namespace Pod Security level.

3.c. Enforce the Pod Security level

When you are satisfied that the chosen level can safely be enforced on the namespace, you can update the namespace to enforce the desired level:

```
kubectl label --overwrite ns $NAMESPACE pod-security.kubernetes.io/enf
```

3.d. Bypass PodSecurityPolicy

Finally, you can effectively bypass PodSecurityPolicy at the namespace level by binding the fully [privileged PSP](#) to all service accounts in the namespace.

```
# The following cluster-scoped commands are only needed once.  
kubectl apply -f privileged-psp.yaml  
kubectl create clusterrole privileged-psp --verb use --resource podsec  
  
# Per-namespace disable  
kubectl create -n $NAMESPACE rolebinding disable-psp --clusterrole pri
```

Since the privileged PSP is non-mutating, and the PSP admission controller always prefers non-mutating PSPs, this will ensure that pods in this namespace are no longer being modified or restricted by PodSecurityPolicy.

The advantage to disabling PodSecurityPolicy on a per-namespace basis like this is if a problem arises you can easily roll the change back by deleting the RoleBinding. Just make sure the pre-existing PodSecurityPolicies are still in place!

```
# Undo PodSecurityPolicy disablement.  
kubectl delete -n $NAMESPACE rolebinding disable-psp
```

4. Review namespace creation processes

Now that existing namespaces have been updated to enforce Pod Security Admission, you should ensure that your processes and/or policies for creating new namespaces are updated to ensure that an appropriate Pod Security profile is applied to new namespaces.

You can also statically configure the Pod Security admission controller to set a default enforce, audit, and/or warn level for unlabeled namespaces. See [Configure the Admission Controller](#) for more information.

5. Disable PodSecurityPolicy

Finally, you're ready to disable PodSecurityPolicy. To do so, you will need to modify the admission configuration of the API server: [How do I turn off an admission controller?](#)

To verify that the PodSecurityPolicy admission controller is no longer enabled, you can manually run a test by impersonating a user without access to any PodSecurityPolicies (see the [PodSecurityPolicy example](#)), or by verifying in the API server logs. At startup, the API server outputs log lines listing the loaded admission controller plugins:

```
I0218 00:59:44.903329      13 plugins.go:158] Loaded 16 mutating admis
I0218 00:59:44.903350      13 plugins.go:161] Loaded 14 validating admis
```

You should see `PodSecurity` (in the validating admission controllers), and neither list should contain `PodSecurityPolicy`.

Once you are certain the PSP admission controller is disabled (and after sufficient soak time to be confident you won't need to roll back), you are free to delete your PodSecurityPolicies and any associated Roles, ClusterRoles, RoleBindings and ClusterRoleBindings (just make sure they don't grant any other unrelated permissions).

4 - Monitoring, Logging, and Debugging

Set up monitoring and logging to troubleshoot a cluster, or debug a containerized application.

Sometimes things go wrong. This guide is aimed at making them right. It has two sections:

- [Debugging your application](#) - Useful for users who are deploying code into Kubernetes and wondering why it is not working.
- [Debugging 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

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 community.

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](#).

Help! My question isn't covered! I need help now!

Stack Exchange, Stack Overflow, or Server Fault

If you have questions related to *software development* for your containerized app, you can ask those on [Stack Overflow](#).

If you have Kubernetes questions related to *cluster management* or *configuration*, you can ask those on [Server Fault](#).

There are also several more specific Stack Exchange network sites which might be the right place to ask Kubernetes questions in areas such as [DevOps](#), [Software Engineering](#), or [InfoSec](#).

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 ensure that your question is on-topic on Stack Overflow, Server Fault, or the Stack Exchange Network site you're asking on**, and read through the guidance on [how to ask a new question](#), before asking a new one!

Slack

Many people from the Kubernetes community hang out on Kubernetes Slack in the `#kubernetes-users` channel. Slack requires registration; you can [request an invitation](#), and registration is open to everyone). Feel free to come and ask any and all questions. Once registered, access the [Kubernetes organisation in Slack](#) via your web browser or via Slack's own dedicated app.

Once you are 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-contributors`](#) channel.

There are also many country specific / local language channels. Feel free to join these channels for localized support and info:

Country	Channels
China	<code>#cn-users</code> , <code>#cn-events</code>
Finland	<code>#fi-users</code>
France	<code>#fr-users</code> , <code>#fr-events</code>
Germany	<code>#de-users</code> , <code>#de-events</code>
India	<code>#in-users</code> , <code>#in-events</code>
Italy	<code>#it-users</code> , <code>#it-events</code>
Japan	<code>#jp-users</code> , <code>#jp-events</code>
Korea	<code>#kr-users</code>
Netherlands	<code>#nl-users</code>
Norway	<code>#norw-users</code>
Poland	<code>#pl-users</code>
Russia	<code>#ru-users</code>
Spain	<code>#es-users</code>
Sweden	<code>#se-users</code>
Turkey	<code>#tr-users</code> , <code>#tr-events</code>

Forum

You're welcome to join the official Kubernetes Forum:
discuss.kubernetes.io.

Bugs and feature requests

If you have what looks like a bug, or you would like to make a feature request, please use the [GitHub issue tracking system](#).

Before you file an issue, please search existing issues to see if your issue is already covered.

If filing a bug, please include detailed information about how to reproduce the problem, such as:

- Kubernetes version: `kubectl version`
- Cloud provider, OS distro, network configuration, and container runtime version
- Steps to reproduce the problem

4.1 - Troubleshooting Applications

Debugging common containerized application issues.

This doc contains a set of resources for fixing issues with containerized applications. It covers things like common issues with Kubernetes resources (like Pods, Services, or StatefulSets), advice on making sense of container termination messages, and ways to debug running containers.

4.1.1 - Debug Pods

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

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 registry?
- Try to manually pull the image to see if the image can be pulled. For example, if you use Docker on your PC, run `docker pull <image>`.

My pod is crashing or otherwise unhealthy

Once your pod has been scheduled, the methods described in [Debug Running Pods](#) are available for debugging.

My pod is running but not doing what I told it to do

If your pod is not behaving as you expected, it may be that there was an error in your pod description (e.g. `mypod.yaml` file on your local machine), and that the error was silently ignored when you created the pod. Often a section of the pod description is nested incorrectly, or a key name is typed incorrectly, and so the key is ignored. For example, if you misspelled `command` as `commnd` then the pod will be created but will not use the command line you intended it to use.

The first thing to do is to delete your pod and try creating it again with the `--validate` option. For example, run `kubectl apply --validate -f mypod.yaml`. If you misspelled `command` as `commnd` then will give an error like this:

```
I0805 10:43:25.129850    46757 schema.go:126] unknown field: commnd
I0805 10:43:25.129973    46757 schema.go:129] this may be a false alarm
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 pods that you expect to be members 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. Verify that the pod's `containerPort` matches up with the Service's `targetPort`.

Network traffic is not forwarded

Please see [debugging service](#) for more information.

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.

4.1.2 - 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 Pods through a Deployment (or other workload controller) 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.

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=gcr.io/google-cont
```

Note: If you don't see a command prompt, try pressing enter.

If you already have a running Pod that you prefer to use, you can run a command in it using:

```
kubectl exec <POD-NAME> -c <CONTAINER-NAME> -- <COMMAND>
```

Setup

For the purposes of this walk-through, let's run some Pods. Since you're probably debugging your own Service you can substitute your own details, or you can follow along and get a second data point.

```
kubectl create deployment hostnames --image=registry.k8s.io/serve_hostname
```

```
deployment.apps/hostnames created
```

`kubectl` commands will print the type and name of the resource created or mutated, which can then be used in subsequent commands.

Let's scale the deployment to 3 replicas.

```
kubectl scale deployment hostnames --replicas=3
```

```
deployment.apps/hostnames scaled
```

Note that this is the same as if you had started the Deployment with the following YAML:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  labels:
    app: hostnames
  name: hostnames
spec:
  selector:
    matchLabels:
      app: hostnames
  replicas: 3
  template:
    metadata:
      labels:
        app: hostnames
    spec:
      containers:
        - name: hostnames
          image: registry.k8s.io/serve_hostname
```

The label "app" is automatically set by `kubectl create deployment` to the name of the Deployment.

You can 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

You can also confirm that your Pods are serving. You can get the list of Pod IP addresses and test them directly.

```
kubectl get pods -l app=hostnames \
-o go-template='{{range .items}}{{.status.podIP}}\n{{end}}'
```

```
10.244.0.5
10.244.0.6
10.244.0.7
```

The example container used for this walk-through serves its own hostname via HTTP on port 9376, but if you are debugging your own app, you'll want to use whatever port number your Pods are listening on.

From within a pod:

```
for ep in 10.244.0.5:9376 10.244.0.6:9376 10.244.0.7:9376; do
  wget -qO- $ep
done
```

This should produce something like:

```
hostnames-632524106-bbpiw  
hostnames-632524106-ly40y  
hostnames-632524106-tlaok
```

If you are not getting the responses you expect at this point, your Pods might not be healthy or might not be listening on the port you think they are. You might find `kubectl logs` to be useful for seeing what is happening, or perhaps you need to `kubectl exec` directly into your Pods and debug from there.

Assuming everything has gone to plan so far, you can start to investigate why your Service doesn't work.

Does the Service exist?

The astute reader will have noticed that you 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.

What would happen if you tried to access a non-existent Service? If you have another Pod that consumes this Service by name you would get something like:

```
wget -O- hostnames
```

```
Resolving hostnames (hostnames)... failed: Name or service not known.  
wget: unable to resolve host address 'hostnames'
```

The first thing to check is whether that Service actually exists:

```
kubectl get svc hostnames
```

```
No resources found.  
Error from server (NotFound): services "hostnames" not found
```

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:

```
kubectl get svc hostnames
```

NAME	TYPE	CLUSTER-IP	EXTERNAL-IP	PORT(S)	AGE
hostnames	ClusterIP	10.0.1.175	<none>	80/TCP	5s

Now you know that the Service exists.

As before, this is the same as if you had started the Service with YAML:

```
apiVersion: v1
kind: Service
metadata:
  labels:
    app: hostnames
    name: hostnames
spec:
  selector:
    app: hostnames
  ports:
  - name: default
    protocol: TCP
    port: 80
    targetPort: 9376
```

In order to highlight the full range of configuration, the Service you created here uses a different port number than the Pods. For many real-world Services, these values might be the same.

Any Network Policy Ingress rules affecting the target Pods?

If you have deployed any Network Policy Ingress rules which may affect incoming traffic to `hostnames-*` Pods, these need to be reviewed.

Please refer to [Network Policies](#) for more details.

Does the Service work by DNS name?

One of the most common ways that clients consume a Service is through a DNS name.

From a Pod in the same Namespace:

```
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 (again, from within a 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:

```
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 you'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 the cluster's DNS Service IP, yours might be different.

```
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 in your Pod is correct. From within a Pod:

```
cat /etc/resolv.conf
```

You should see something like:

```
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 lastly for names in 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. Throughout this document, the cluster suffix is assumed to be "cluster.local". Your own clusters might be configured differently, in which case you should change that in all of the previous commands.

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 work by DNS name?

If the above still fails, DNS lookups are not working for your Service. You can take a step back and see what else is not working. The Kubernetes master Service should always work. From within a 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, please see the [kube-proxy](#) section of this document, or even go back to the top of this document and start over, but instead of debugging your own Service, debug the DNS Service.

Does the Service work by IP?

Assuming you have confirmed that DNS works, the next thing to test is whether your Service works by its IP address. From a Pod in your cluster, access the Service's IP (from `kubectl get` above).

```
for i in $(seq 1 3); do
    wget -qO- 10.0.1.175:80
done
```

This should produce something like:

```
hostnames-632524106-bbpiw
hostnames-632524106-ly40y
hostnames-632524106-tlaok
```

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 defined correctly?

It might sound silly, but you should really double and triple check that your Service is correct and matches your Pod's port. Read back your Service and verify it:

```
kubectl get service hostnames -o json
```

```
{
  "kind": "Service",
  "apiVersion": "v1",
  "metadata": {
    "name": "hostnames",
    "namespace": "default",
    "uid": "428c8b6c-24bc-11e5-936d-42010af0a9bc",
    "resourceVersion": "347189",
    "creationTimestamp": "2015-07-07T15:24:29Z",
    "labels": {
      "app": "hostnames"
    }
  },
  "spec": {
    "ports": [
      {
        "name": "default",
        "protocol": "TCP",
        "port": 80,
        "targetPort": 9376,
        "nodePort": 0
      }
    ],
    "selector": {
      "app": "hostnames"
    },
    "clusterIP": "10.0.1.175",
    "type": "ClusterIP",
    "sessionAffinity": "None"
  },
  "status": {
    "loadBalancer": {}
  }
}
```

- Is the Service port you are trying to access listed in `spec.ports[]` ?
- Is the `targetPort` correct for your Pods (some Pods use a different port than the Service)?
- If you meant to use a numeric port, is it a number (9376) or a string "9376"?
- If you meant to use a named port, do your Pods expose a port with the same name?
- Is the port's `protocol` correct for your Pods?

Does the Service have any Endpoints?

If you got this far, you have confirmed that your Service is correctly defined and is resolved by DNS. Now let's check that the Pods you ran are actually being selected by the Service.

Earlier you saw that the Pods were running. You can re-check that:

```
kubectl get pods -l app=hostnames
```

NAME	READY	STATUS	RESTARTS	AGE
hostnames-632524106-bbpiw	1/1	Running	0	1h
hostnames-632524106-ly40y	1/1	Running	0	1h
hostnames-632524106-tlaok	1/1	Running	0	1h

The `-l app=hostnames` argument is a label selector configured on the Service.

The "AGE" column says that these Pods are about an hour old, which implies that they are running fine and not crashing.

The "RESTARTS" column says that these pods are not crashing frequently or being restarted. Frequent restarts could lead to intermittent connectivity issues. If the restart count is high, read more about how to [debug pods](#).

Inside the Kubernetes system is a control loop which evaluates the selector of every Service and saves the results into a corresponding 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 `ENDPOINTS` column is `<none>`, 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 `app=hostnames`, but the Deployment specifying `run=hostnames`, as in versions previous to 1.18, where the `kubectl run` command could have been also used to create a Deployment.

Are the Pods working?

At this point, you know that your Service exists and has selected your Pods. At the beginning of this walk-through, you verified the Pods themselves. Let's check again that the Pods are actually working - you can bypass the Service mechanism and go straight to the Pods, as listed by the Endpoints above.

Note: These commands use the Pod port (9376), rather than the Service port (80).

From within a Pod:

```
for ep in 10.244.0.5:9376 10.244.0.6:9376 10.244.0.7:9376; do  
  wget -qO- $ep  
done
```

This should produce something like:

```
hostnames-632524106-bbpiw  
hostnames-632524106-ly40y  
hostnames-632524106-tlaok
```

You 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.

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.

The default implementation of Services, and the one used on most clusters, is kube-proxy. This is a program that runs on every node and configures one of a small set of mechanisms for providing the Service abstraction. If your cluster does not use kube-proxy, the following sections will not apply, and you will have to investigate whatever implementation of Services you are using.

Is kube-proxy running?

Confirm that `kube-proxy` is running on your Nodes. Running directly on a Node, you should get something like the below:

```
ps auxw | grep kube-proxy
```

```
root  4194  0.4  0.1 101864 17696 ?      S1 Jul04  25:43 /usr/local/bin/k
```

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 mode
I1027 22:14:53.998163    5063 server.go:247] Using iptables Proxier.
I1027 22:14:54.038140    5063 proxier.go:352] Setting endpoints for "kubernetes"
I1027 22:14:54.038164    5063 proxier.go:352] Setting endpoints for "kubernetes"
I1027 22:14:54.038209    5063 proxier.go:352] Setting endpoints for "default"
I1027 22:14:54.038238    5063 proxier.go:429] Not syncing iptables until
I1027 22:14:54.040048    5063 proxier.go:294] Adding new service "default"
I1027 22:14:54.040154    5063 proxier.go:294] Adding new service "kube-apiserver"
I1027 22:14:54.040223    5063 proxier.go:294] Adding new service "kube-
```

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.

Kube-proxy can run in one of a few modes. In the log listed above, the line `Using iptables Proxier` indicates that kube-proxy is running in "iptables" mode. The most common other mode is "ipvs".

Iptables mode

In "iptables" mode, you should see something like the following on a Node:

```
iptables-save | grep hostnames
```

```
-A KUBE-SEP-57KPRZ3JQVENLNBR -s 10.244.3.6/32 -m comment --comment "def
-A KUBE-SEP-57KPRZ3JQVENLNBR -p tcp -m comment --comment "default/hostn
-A KUBE-SEP-WNBA2IHDGP2BOBGZ -s 10.244.1.7/32 -m comment --comment "def
-A KUBE-SEP-WNBA2IHDGP2BOBGZ -p tcp -m comment --comment "default/hostn
-A KUBE-SEP-X3P2623AGDH6CDF3 -s 10.244.2.3/32 -m comment --comment "def
-A KUBE-SEP-X3P2623AGDH6CDF3 -p tcp -m comment --comment "default/hostn
-A KUBE-SERVICES -d 10.0.1.175/32 -p tcp -m comment --comment "default/
-A KUBE-SVC-NWV5X2332I4OT4T3 -m comment --comment "default/hostnames:"
-A KUBE-SVC-NWV5X2332I4OT4T3 -m comment --comment "default/hostnames:"
-A KUBE-SVC-NWV5X2332I4OT4T3 -m comment --comment "default/hostnames:"
```

For each port of each Service, there should be 1 rule in `KUBE-SERVICES` and one `KUBE-SVC-<hash>` chain. For each Pod endpoint, there should be a small number of rules in that `KUBE-SVC-<hash>` and one `KUBE-SEP-<hash>` chain with a small number of rules in it. The exact rules will vary based on your exact config (including node-ports and load-balancers).

IPVS mode

In "ipvs" mode, you should see something like the following on a Node:

```
ipvsadm -ln
```

Prot LocalAddress:Port Scheduler Flags		Forward	Weight	ActiveConn	InActConn
-> RemoteAddress:Port					
...					
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	
...					

For each port of each Service, plus any NodePorts, external IPs, and load-balancer IPs, kube-proxy will create a virtual server. For each Pod endpoint, it will create corresponding real servers. 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`).

Is kube-proxy proxying?

Assuming you do see one the above cases, try again to access your Service by IP from one of your Nodes:

```
curl 10.0.1.175:80
```

```
hostnames-632524106-bbpiw
```

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.
```

If you don't see those, try restarting `kube-proxy` with the `-v` flag set to 4, and then look at the logs again.

Edge case: A Pod fails to reach itself via the Service IP

This might sound unlikely, but it does happen and it is supposed to work.

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.

```
ps auxw | grep kubelet
```

```
root      3392  1.1  0.8 186804 65208 ?          S1   00:51  11:11 /usr/1
```

- 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 "prom
```

- 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
```

```
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:

```
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, and `kube-proxy` does not seem to be misbehaving. And yet your Service is not working. Please let us know what is going on, so we can help investigate!

Contact us on [Slack](#) or [Forum](#) or [GitHub](#).

What's next

Visit the [troubleshooting overview document](#) for more information.

4.1.3 - Debug a StatefulSet

This task shows you how to debug a StatefulSet.

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.kubernetes.io/name=MyApp` set on them, you can use the following:

```
kubectl get pods -l app.kubernetes.io/name=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](#).

4.1.4 - 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](#).

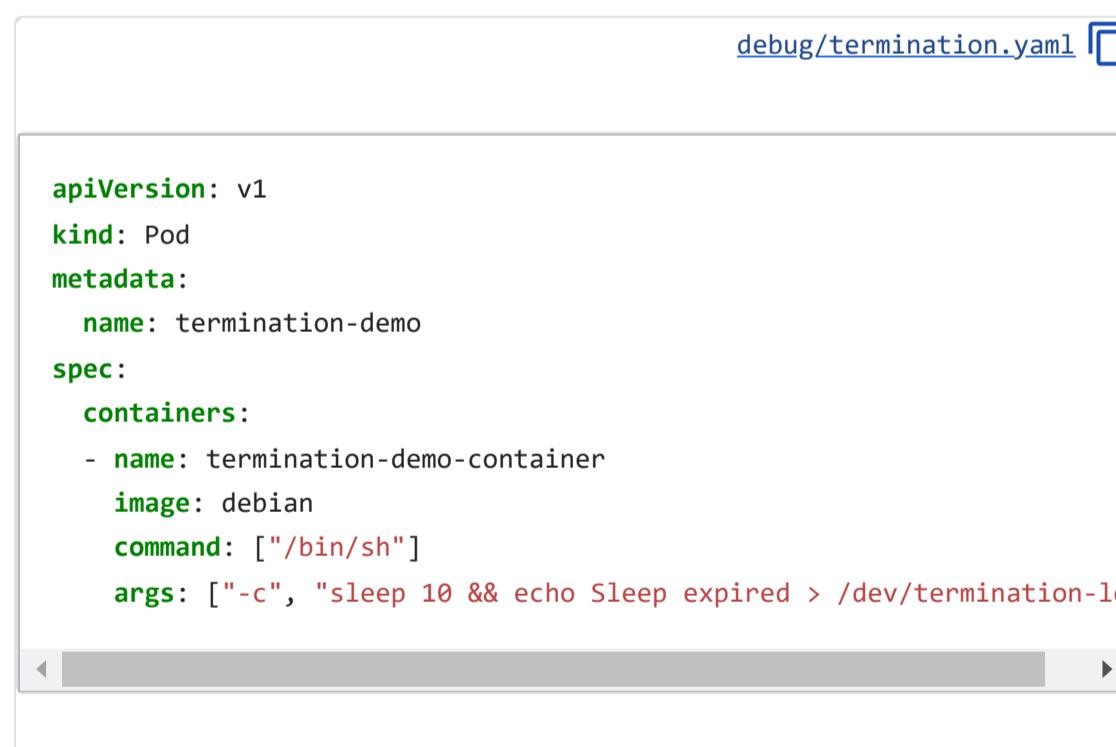
Before you begin

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

- [Killercoda](#)
- [Play with Kubernetes](#)

Writing and reading a termination message

In this exercise, you create a Pod that runs one container. The manifest for that Pod specifies a command that runs when the container starts:



```
apiVersion: v1
kind: Pod
metadata:
  name: termination-demo
spec:
  containers:
    - name: termination-demo-container
      image: debian
      command: ["/bin/sh"]
      args: ["-c", "sleep 10 && echo Sleep expired > /dev/termination-log"]
```

1. Create a Pod based on the YAML configuration file:

```
kubectl apply -f https://k8s.io/examples/debug/termination.yaml
```

In the YAML file, in the `command` and `args` fields, you can see that the container sleeps for 10 seconds and then writes "Sleep expired" to the `/dev/termination-log` file. After the container writes the "Sleep expired" message, it terminates.

2. Display information about the Pod:

```
kubectl get pod termination-demo
```

Repeat the preceding command until the Pod is no longer running.

3. Display detailed information about the Pod:

```
kubectl get pod termination-demo --output=yaml
```

The output includes the "Sleep expired" message:

```
apiVersion: v1
kind: Pod
...
lastState:
  terminated:
    containerID: ...
    exitCode: 0
    finishedAt: ...
    message: |
      Sleep expired
...
```

4. Use a Go template to filter the output so that it includes only the termination message:

```
kubectl get pod termination-demo -o go-template="{{range .status.
```

If you are running a multi-container Pod, you can use a Go template to include the container's name. By doing so, you can discover which of the containers is failing:

```
kubectl get pod multi-container-pod -o go-template='{{range .status.co
```

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.

The termination message is intended to be brief final status, such as an assertion failure message. The kubelet truncates messages that are longer than 4096 bytes.

The total message length across all containers is limited to 12KiB, divided equally among each container. For example, if there are 12 containers (`initContainers` or `containers`), each has 1024 bytes of available termination message space.

The default termination message path is `/dev/termination-log`. You cannot set the termination message path after a Pod is launched.

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](#).

4.1.5 - 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

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

- [KillerCoda](#)
- [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
<code><pod-name></code>	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 <code>M</code> Init Containers, and <code>N</code> have completed so far.
<code>Init:Error</code>	An Init Container has failed to execute.
<code>Init:CrashLoopBackOff</code>	An Init Container has failed repeatedly.

Status	Meaning
Pending	The Pod has not yet begun executing Init Containers.
PodInitializing or Running	The Pod has already finished executing Init Containers.

4.1.6 - Debug Running Pods

This page explains how to debug Pods running (or crashing) on a Node.

Before you begin

- Your Pod should already be scheduled and running. If your Pod is not yet running, start with [Debugging Pods](#).
- For some of the advanced debugging steps you need to know on which Node the Pod is running and have shell access to run commands on that Node. You don't need that access to run the standard debug steps that use `kubectl`.

Using `kubectl describe pod` to fetch details about pods

For this example we'll use a Deployment to create two pods, similar to the earlier example.

```
application/nginx-with-request.yaml □
```

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-deployment
spec:
  selector:
    matchLabels:
      app: nginx
  replicas: 2
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
        - name: nginx
          image: nginx
          resources:
            limits:
              memory: "128Mi"
              cpu: "500m"
          ports:
            - containerPort: 80
```

Create deployment by running following command:

```
kubectl apply -f https://k8s.io/examples/application/nginx-with-request.yaml
```

```
deployment.apps/nginx-deployment created
```

Check pod status by following command:

```
kubectl get pods
```

NAME	READY	STATUS	RESTARTS	AGE
nginx-deployment-67d4bdd6f5-cx2nz	1/1	Running	0	13s
nginx-deployment-67d4bdd6f5-w6kd7	1/1	Running	0	13s

We can retrieve a lot more information about each of these pods using `kubectl describe pod`. For example:

```
kubectl describe pod nginx-deployment-67d4bdd6f5-w6kd7
```

```

Name: nginx-deployment-67d4bdd6f5-w6kd7
Namespace: default
Priority: 0
Node: kube-worker-1/192.168.0.113
Start Time: Thu, 17 Feb 2022 16:51:01 -0500
Labels: app=nginx
        pod-template-hash=67d4bdd6f5
Annotations: <none>
Status: Running
IP: 10.88.0.3
IPs:
  IP: 10.88.0.3
  IP: 2001:db8::1
Controlled By: ReplicaSet/nginx-deployment-67d4bdd6f5
Containers:
  nginx:
    Container ID: containerd://5403af59a2b46ee5a23fb0ae4b1e077f7ca5c5
    Image: nginx
    Image ID: docker.io/library/nginx@sha256:2834dc507516af027848
    Port: 80/TCP
    Host Port: 0/TCP
    State: Running
    Started: Thu, 17 Feb 2022 16:51:05 -0500
    Ready: True
    Restart Count: 0
    Limits:
      cpu: 500m
      memory: 128Mi
    Requests:
      cpu: 500m
      memory: 128Mi
    Environment: <none>
    Mounts:
      /var/run/secrets/kubernetes.io/serviceaccount from kube-api-access (ro)
Conditions:
  Type Status
  Initialized True
  Ready True
  ContainersReady True
  PodScheduled True
Volumes:
  kube-api-access-bgsgp:
    Type: Projected (a volume that contains injected tokens)
    TokenExpirationSeconds: 3607
    ConfigMapName: kube-root-ca.crt
    ConfigMapOptional: <nil>
    DownwardAPI: true
    QoS Class: Guaranteed
    Node-Selectors: <none>
    Tolerations: node.kubernetes.io/not-ready:NoExecute op=
                  node.kubernetes.io/unreachable:NoExecute o
Events:
  Type Reason Age From Message
  ---- ---- - - -
  Normal Scheduled 34s default-scheduler Successfully assigned def
  Normal Pulling 31s kubelet Pulling image "nginx"
  Normal Pulled 30s kubelet Successfully pulled image
  Normal Created 30s kubelet Created container nginx
  Normal Started 30s kubelet Started container nginx

```

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. "From" indicates the component that is logging the event. "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
  -----  -----  -----
  1m        48s       7  {default-scheduler }
  fit failure on node (kubernetes-node-6ta5): Node didn't have enough resources
  fit failure on node (kubernetes-node-wul5): Node didn't have enough resources
```

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 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:
  creationTimestamp: "2022-02-17T21:51:01Z"
  generateName: nginx-deployment-67d4bdd6f5-
  labels:
    app: nginx
    pod-template-hash: 67d4bdd6f5
    name: nginx-deployment-67d4bdd6f5-w6kd7
  namespace: default
  ownerReferences:
    - apiVersion: apps/v1
      blockOwnerDeletion: true
      controller: true
      kind: ReplicaSet
      name: nginx-deployment-67d4bdd6f5
      uid: 7d41dfd4-84c0-4be4-88ab-cedbe626ad82
  resourceVersion: "1364"
  uid: a6501da1-0447-4262-98eb-c03d4002222e
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
      terminationMessagePolicy: File
      volumeMounts:
        - mountPath: /var/run/secrets/kubernetes.io/serviceaccount
          name: kube-api-access-bgsgp
          readOnly: true
    dnsPolicy: ClusterFirst
    enableServiceLinks: true
    nodeName: kube-worker-1
    preemptionPolicy: PreemptLowerPriority
    priority: 0
    restartPolicy: Always
    schedulerName: default-scheduler
    securityContext: {}
    serviceAccount: default
    serviceAccountName: default
    terminationGracePeriodSeconds: 30
    tolerations:
      - effect: NoExecute
        key: node.kubernetes.io/not-ready
        operator: Exists
        tolerationSeconds: 300
      - effect: NoExecute
        key: node.kubernetes.io/unreachable
        operator: Exists
        tolerationSeconds: 300
    volumes:
      - name: kube-api-access-bgsgp
    projected:
      defaultMode: 420
      sources:
        - serviceAccountToken:
```

```
expirationSeconds: 3607
path: token
- configMap:
  items:
  - key: ca.crt
    path: ca.crt
    name: kube-root-ca.crt
- downwardAPI:
  items:
  - fieldRef:
    apiVersion: v1
    fieldPath: metadata.namespace
    path: namespace
status:
  conditions:
  - lastProbeTime: null
    lastTransitionTime: "2022-02-17T21:51:01Z"
    status: "True"
    type: Initialized
  - lastProbeTime: null
    lastTransitionTime: "2022-02-17T21:51:06Z"
    status: "True"
    type: Ready
  - lastProbeTime: null
    lastTransitionTime: "2022-02-17T21:51:06Z"
    status: "True"
    type: ContainersReady
  - lastProbeTime: null
    lastTransitionTime: "2022-02-17T21:51:01Z"
    status: "True"
    type: PodScheduled
  containerStatuses:
  - containerID: containerd://5403af59a2b46ee5a23fb0ae4b1e077f7ca5c5fb
    image: docker.io/library/nginx:latest
    imageID: docker.io/library/nginx@sha256:2834dc507516af02784808c5f4
    lastState: {}
    name: nginx
    ready: true
    restartCount: 0
    started: true
    state:
      running:
        startedAt: "2022-02-17T21:51:05Z"
  hostIP: 192.168.0.113
  phase: Running
  podIP: 10.88.0.3
  podIPs:
  - ip: 10.88.0.3
  - ip: 2001:db8::1
  qosClass: Guaranteed
  startTime: "2022-02-17T21:51:01Z"
```

Examining pod logs

First, look at the logs of the affected 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}
```

Debugging with container exec

If the container image includes debugging utilities, as is the case with images built from Linux and Windows OS base images, you can run commands inside a specific container with `kubectl exec`:

```
kubectl exec ${POD_NAME} -c ${CONTAINER_NAME} -- ${CMD} ${ARG1} ${ARG2}
```

Note: `-c ${CONTAINER_NAME}` is optional. You can omit it for Pods that only contain a single container.

As an example, to look at the logs from a running Cassandra pod, you might run

```
kubectl exec cassandra -- cat /var/log/cassandra/system.log
```

You can run a shell that's connected to your terminal using the `-i` and `-t` arguments to `kubectl exec`, for example:

```
kubectl exec -it cassandra -- sh
```

For more details, see [Get a Shell to a Running Container](#).

Debugging with an ephemeral debug container

FEATURE STATE: [Kubernetes v1.25 \[stable\]](#)

Ephemeral containers are useful for interactive troubleshooting when `kubectl exec` is insufficient because a container has crashed or a container image doesn't include debugging utilities, such as with [distroless images](#).

Example debugging using ephemeral containers

You can use the `kubectl debug` command to add ephemeral containers to a running Pod. First, create a pod for the example:

```
kubectl run ephemeral-demo --image=registry.k8s.io/pause:3.1 --restart
```

The examples in this section use the `pause` container image because it does not contain debugging utilities, but this method works with all container images.

If you attempt to use `kubectl exec` to create a shell you will see an error because there is no shell in this container image.

```
kubectl exec -it ephemeral-demo -- sh
```

```
OCI runtime exec failed: exec failed: container_linux.go:346: starting
```

You can instead add a debugging container using `kubectl debug`. If you specify the `-i` / `--interactive` argument, `kubectl` will automatically attach to the console of the Ephemeral Container.

```
kubectl debug -it ephemeral-demo --image=busybox:1.28 --target=ephemeral
```

```
Defaulting debug container name to debugger-8xzrl.  
If you don't see a command prompt, try pressing enter.  
/ #
```

This command adds a new busybox container and attaches to it. The `--target` parameter targets the process namespace of another container. It's necessary here because `kubectl run` does not enable [process namespace sharing](#) in the pod it creates.

Note: The `--target` parameter must be supported by the Container Runtime. When not supported, the Ephemeral Container may not be started, or it may be started with an isolated process namespace so that `ps` does not reveal processes in other containers.

You can view the state of the newly created ephemeral container using `kubectl describe`:

```
kubectl describe pod ephemeral-demo
```

```
...  
Ephemeral Containers:  
debugger-8xzrl:  
  Container ID: docker://b888f9adfd15bd5739fefaa39e1df4dd3c617b9902  
  Image:        busybox  
  Image ID:     docker-pullable://busybox@sha256:1828edd60c5efd34b2  
  Port:          <none>  
  Host Port:    <none>  
  State:         Running  
  Started:      Wed, 12 Feb 2020 14:25:42 +0100  
  Ready:         False  
  Restart Count: 0  
  Environment:  <none>  
  Mounts:        <none>  
...
```

Use `kubectl delete` to remove the Pod when you're finished:

```
kubectl delete pod ephemeral-demo
```

Debugging using a copy of the Pod

Sometimes Pod configuration options make it difficult to troubleshoot in certain situations. For example, you can't run `kubectl exec` to troubleshoot your container if your container image does not include a shell or if your application crashes on startup. In these situations you can use `kubectl debug` to create a copy of the Pod with configuration values changed to aid debugging.

Copying a Pod while adding a new container

Adding a new container can be useful when your application is running but not behaving as you expect and you'd like to add additional troubleshooting utilities to the Pod.

For example, maybe your application's container images are built on `busybox` but you need debugging utilities not included in `busybox`. You can simulate this scenario using `kubectl run`:

```
kubectl run myapp --image=busybox:1.28 --restart=Never -- sleep 1d
```

Run this command to create a copy of `myapp` named `myapp-debug` that adds a new Ubuntu container for debugging:

```
kubectl debug myapp -it --image=ubuntu --share-processes --copy-to=myap
```

```
Defaulting debug container name to debugger-w7xmf.  
If you don't see a command prompt, try pressing enter.  
root@myapp-debug:/#
```

Note:

- `kubectl debug` automatically generates a container name if you don't choose one using the `--container` flag.
- The `-i` flag causes `kubectl debug` to attach to the new container by default. You can prevent this by specifying `--attach=false`. If your session becomes disconnected you can reattach using `kubectl attach`.
- The `--share-processes` allows the containers in this Pod to see processes from the other containers in the Pod. For more information about how this works, see [Share Process Namespace between Containers in a Pod](#).

Don't forget to clean up the debugging Pod when you're finished with it:

```
kubectl delete pod myapp myapp-debug
```

Copying a Pod while changing its command

Sometimes it's useful to change the command for a container, for example to add a debugging flag or because the application is crashing.

To simulate a crashing application, use `kubectl run` to create a container that immediately exits:

```
kubectl run --image=busybox:1.28 myapp -- false
```

You can see using `kubectl describe pod myapp` that this container is crashing:

```
Containers:  
myapp:  
  Image:      busybox  
  ...  
  Args:  
    false  
  State:      Waiting  
    Reason:    CrashLoopBackOff  
  Last State: Terminated  
    Reason:    Error  
    Exit Code: 1
```

You can use `kubectl debug` to create a copy of this Pod with the command changed to an interactive shell:

```
kubectl debug myapp -it --copy-to=myapp-debug --container=myapp -- sh
```

```
◀ ━━━━ ▶  
If you don't see a command prompt, try pressing enter.  
/ #
```

Now you have an interactive shell that you can use to perform tasks like checking filesystem paths or running the container command manually.

Note:

- To change the command of a specific container you must specify its name using `--container` or `kubectl debug` will instead create a new container to run the command you specified.
- The `-i` flag causes `kubectl debug` to attach to the container by default. You can prevent this by specifying `--attach=false`. If your session becomes disconnected you can reattach using `kubectl attach`.

Don't forget to clean up the debugging Pod when you're finished with it:

```
kubectl delete pod myapp myapp-debug
```

Copying a Pod while changing container images

In some situations you may want to change a misbehaving Pod from its normal production container images to an image containing a debugging build or additional utilities.

As an example, create a Pod using `kubectl run`:

```
kubectl run myapp --image=busybox:1.28 --restart=Never -- sleep 1d
```

Now use `kubectl debug` to make a copy and change its container image to `ubuntu`:

```
kubectl debug myapp --copy-to=myapp-debug --set-image*=ubuntu
```

The syntax of `--set-image` uses the same `container_name=image` syntax as `kubectl set image . *=ubuntu` means change the image of all containers to `ubuntu`.

Don't forget to clean up the debugging Pod when you're finished with it:

```
kubectl delete pod myapp myapp-debug
```

Debugging via a shell on the node

If none of these approaches work, you can find the Node on which the Pod is running and create a Pod running on the Node. To create an interactive shell on a Node using `kubectl debug`, run:

```
kubectl debug node/mynode -it --image=ubuntu
```

```
Creating debugging pod node-debugger-mynode-pdx84 with container debugger
If you don't see a command prompt, try pressing enter.
root@ek8s:/#
```

When creating a debugging session on a node, keep in mind that:

- `kubectl debug` automatically generates the name of the new Pod based on the name of the Node.
- The root filesystem of the Node will be mounted at `/host`.
- The container runs in the host IPC, Network, and PID namespaces, although the pod isn't privileged, so reading some process information may fail, and `chroot /host` may fail.
- If you need a privileged pod, create it manually.

Don't forget to clean up the debugging Pod when you're finished with it:

```
kubectl delete pod node-debugger-mynode-pdx84
```

4.1.7 - 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

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

- [Killercoda](#)
- [Play with Kubernetes](#)

Getting a shell to a container

In this exercise, you create a Pod that has one container. The container runs the nginx image. Here is the configuration file for the Pod:

```
application/shell-demo.yaml 
```

```
apiVersion: v1
kind: Pod
metadata:
  name: shell-demo
spec:
  volumes:
    - name: shared-data
      emptyDir: {}
  containers:
    - name: nginx
      image: nginx
      volumeMounts:
        - name: shared-data
          mountPath: /usr/share/nginx/html
  hostNetwork: true
  dnsPolicy: Default
```

Create the Pod:

```
kubectl apply -f https://k8s.io/examples/application/shell-demo.yaml
```

Verify that the container is running:

```
kubectl get pod shell-demo
```

Get a shell to the running container:

```
kubectl exec --stdin --tty shell-demo -- /bin/bash
```

Note: The double dash (--) separates the arguments you want to pass to the command from the kubectl arguments.

In your shell, list the root directory:

```
# Run this inside the container
ls /
```

In your shell, experiment with other commands. Here are some examples:

```
# You can run these example commands inside the container
ls /
cat /proc/mounts
cat /proc/1/maps
apt-get update
apt-get install -y tcpdump
tcpdump
apt-get install -y lsof
lsof
apt-get install -y procps
ps aux
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:

```
# Run this inside the container
echo 'Hello shell demo' > /usr/share/nginx/html/index.html
```

In your shell, send a GET request to the nginx server:

```
# Run this in the shell inside your container
apt-get update
apt-get install curl
curl http://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`.

```
exit # To quit the shell in the container
```

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 with 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 -i -t my-pod --container main-app -- /bin/bash
```

Note: The short options `-i` and `-t` are the same as the long options `--stdin` and `--tty`

What's next

- Read about [kubectl exec](#)

4.2 - Troubleshooting Clusters

Debugging common cluster issues.

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 the [troubleshooting overview document](#) for more information.

For troubleshooting kubectl, refer to [Troubleshooting kubectl](#).

List your cluster

The first thing to debug in your cluster is if your nodes are all registered correctly.

Run the following command:

```
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.

To get detailed information about the overall health of your cluster, you can run:

```
kubectl cluster-info dump
```

Example: debugging a down/unreachable node

Sometimes when debugging it can be useful to look at the status of a node -- for example, because you've noticed strange behavior of a Pod that's running on the node, or to find out why a Pod won't schedule onto the node. As with Pods, you can use `kubectl describe node` and `kubectl get node -o yaml` to retrieve detailed information about nodes. For example, here's what you'll see if a node is down (disconnected from the network, or kubelet dies and won't restart, etc.). Notice the events that show the node is NotReady, and also notice that the pods are no longer running (they are evicted after five minutes of NotReady status).

```
kubectl get nodes
```

NAME	STATUS	ROLES	AGE	VERSION
kube-worker-1	NotReady	<none>	1h	v1.23.3
kubernetes-node-bols	Ready	<none>	1h	v1.23.3
kubernetes-node-st6x	Ready	<none>	1h	v1.23.3
kubernetes-node-unaj	Ready	<none>	1h	v1.23.3

```
kubectl describe node kube-worker-1
```

Name: kube-worker-1
Roles: <none>
Labels: beta.kubernetes.io/arch=amd64
beta.kubernetes.io/os=linux
kubernetes.io/arch=amd64
kubernetes.io/hostname=kube-worker-1
kubernetes.io/os=linux
Annotations: kubeadm.alpha.kubernetes.io/cri-socket: /run/contai
node.alpha.kubernetes.io/ttl: 0
volumes.kubernetes.io/controller-managed-attach-det
CreationTimestamp: Thu, 17 Feb 2022 16:46:30 -0500
Taints: node.kubernetes.io/unreachable:NoExecute
node.kubernetes.io/unreachable:NoSchedule
Unschedulable: false
Lease:
HolderIdentity: kube-worker-1
AcquireTime: <unset>
RenewTime: Thu, 17 Feb 2022 17:13:09 -0500
Conditions:

Type	Status	LastHeartbeatTime	LastTransitionTime
NetworkUnavailable	False	Thu, 17 Feb 2022 17:09:13 -0500	Thu, 17 Feb 2022 16:46:30 -0500
MemoryPressure	Unknown	Thu, 17 Feb 2022 17:12:40 -0500	Thu, 17 Feb 2022 16:46:30 -0500
DiskPressure	Unknown	Thu, 17 Feb 2022 17:12:40 -0500	Thu, 17 Feb 2022 16:46:30 -0500
PIDPressure	Unknown	Thu, 17 Feb 2022 17:12:40 -0500	Thu, 17 Feb 2022 16:46:30 -0500
Ready	Unknown	Thu, 17 Feb 2022 17:12:40 -0500	Thu, 17 Feb 2022 16:46:30 -0500

Addresses:
InternalIP: 192.168.0.113
Hostname: kube-worker-1
Capacity:
cpu: 2
ephemeral-storage: 15372232Ki
hugepages-2Mi: 0
memory: 2025188Ki
pods: 110
Allocatable:
cpu: 2
ephemeral-storage: 14167048988
hugepages-2Mi: 0
memory: 1922788Ki
pods: 110
System Info:
Machine ID: 9384e2927f544209b5d7b67474bbf92b
System UUID: aa829ca9-73d7-064d-9019-df07404ad448
Boot ID: 5a295a03-aaca-4340-af20-1327fa5dab5c
Kernel Version: 5.13.0-28-generic
OS Image: Ubuntu 21.10
Operating System: linux
Architecture: amd64
Container Runtime Version: containerd://1.5.9
Kubelet Version: v1.23.3
Kube-Proxy Version: v1.23.3
Non-terminated Pods: (4 in total)

Namespace	Name	CPU	Memory
default	nginx-deployment-67d4bdd6f5-cx2nz	500m (5%)	100Mi (13%)
default	nginx-deployment-67d4bdd6f5-w6kd7	500m (5%)	100Mi (13%)
kube-system	kube-proxy-dnxbz	0 (0%)	100Mi (13%)
kube-system	weave-net-gjxxp	100m (2%)	100Mi (13%)

Allocated resources:
(Total limits may be over 100 percent, i.e., overcommitted.)

Resource	Requests	Limits
cpu	1100m (55%)	1 (50%)
memory	456Mi (24%)	256Mi (13%)
ephemeral-storage	0 (0%)	0 (0%)
hugepages-2Mi	0 (0%)	0 (0%)

Events:
...

```
kubectl get node kube-worker-1 -o yaml
```

```
apiVersion: v1
kind: Node
metadata:
  annotations:
    kubeadm.alpha.kubernetes.io/cri-socket: /run/containerd/containerd
    node.alpha.kubernetes.io/ttl: "0"
    volumes.kubernetes.io/controller-managed-attach-detach: "true"
  creationTimestamp: "2022-02-17T21:46:30Z"
  labels:
    beta.kubernetes.io/arch: amd64
    beta.kubernetes.io/os: linux
    kubernetes.io/arch: amd64
    kubernetes.io/hostname: kube-worker-1
    kubernetes.io/os: linux
  name: kube-worker-1
  resourceVersion: "4026"
  uid: 98efe7cb-2978-4a0b-842a-1a7bf12c05f8
spec: {}
status:
  addresses:
  - address: 192.168.0.113
    type: InternalIP
  - address: kube-worker-1
    type: Hostname
  allocatable:
    cpu: "2"
    ephemeral-storage: "14167048988"
    hugepages-2Mi: "0"
    memory: 1922788Ki
    pods: "110"
  capacity:
    cpu: "2"
    ephemeral-storage: 15372232Ki
    hugepages-2Mi: "0"
    memory: 2025188Ki
    pods: "110"
  conditions:
  - lastHeartbeatTime: "2022-02-17T22:20:32Z"
    lastTransitionTime: "2022-02-17T22:20:32Z"
    message: Weave pod has set this
    reason: WeaveIsUp
    status: "False"
    type: NetworkUnavailable
  - lastHeartbeatTime: "2022-02-17T22:20:15Z"
    lastTransitionTime: "2022-02-17T22:13:25Z"
    message: kubelet has sufficient memory available
    reason: KubeletHasSufficientMemory
    status: "False"
    type: MemoryPressure
  - lastHeartbeatTime: "2022-02-17T22:20:15Z"
    lastTransitionTime: "2022-02-17T22:13:25Z"
    message: kubelet has no disk pressure
    reason: KubeletHasNoDiskPressure
    status: "False"
    type: DiskPressure
  - lastHeartbeatTime: "2022-02-17T22:20:15Z"
    lastTransitionTime: "2022-02-17T22:13:25Z"
    message: kubelet has sufficient PID available
    reason: KubeletHasSufficientPID
    status: "False"
    type: PIDPressure
  - lastHeartbeatTime: "2022-02-17T22:20:15Z"
    lastTransitionTime: "2022-02-17T22:15:15Z"
    message: kubelet is posting ready status. AppArmor enabled
    reason: KubeletReady
    status: "True"
```

```
type: Ready
daemonEndpoints:
  kubeletEndpoint:
    Port: 10250
nodeInfo:
  architecture: amd64
  bootID: 22333234-7a6b-44d4-9ce1-67e31dc7e369
  containerRuntimeVersion: containerd://1.5.9
  kernelVersion: 5.13.0-28-generic
  kubeProxyVersion: v1.23.3
  kubeletVersion: v1.23.3
  machineID: 9384e2927f544209b5d7b67474bbf92b
  operatingSystem: linux
  osImage: Ubuntu 21.10
  systemUUID: aa829ca9-73d7-064d-9019-df07404ad448
```

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. On systemd-based systems, you may need to use `journalctl` instead of examining log files.

Control Plane nodes

- `/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` - a component that runs most Kubernetes built-in controllers, with the notable exception of scheduling (the kube-scheduler handles scheduling).

Worker Nodes

- `/var/log/kubelet.log` - logs from the kubelet, responsible for running containers on the node
- `/var/log/kube-proxy.log` - logs from `kube-proxy`, which is responsible for directing traffic to Service endpoints

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.

Contributing 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, for example misconfigured Kubernetes software or application software

Specific scenarios

- API server 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
- API server backing storage lost
 - Results
 - the kube-apiserver component fails to start successfully and become healthy
 - kubelets will not be able to reach it but will continue to run the same pods and provide the same service proxying
 - manual recovery or recreation of apiserver state necessary before apiserver is restarted
- Supporting services (node controller, replication controller manager, scheduler, etc) VM shutdown or crashes
 - currently those are colocated with the apiserver, and their unavailability has similar consequences as apiserver
 - in future, these will be replicated as well and may not be co-located
 - they do not have their own persistent state
- Individual node (VM or physical machine) shuts down
 - Results
 - pods on that Node stop running
- Network partition
 - Results
 - partition A thinks the nodes in partition B are down; partition B thinks the apiserver is down. (Assuming the master VM ends up in partition A.)
- Kubelet software fault
 - Results
 - crashing kubelet cannot start new pods on the node
 - kubelet might delete the pods or not
 - node marked unhealthy
 - replication controllers start new pods elsewhere
- Cluster operator error
 - Results
 - loss of pods, services, etc
 - lost of apiserver backing store
 - users unable to read API
 - etc.

Mitigations

- Action: Use IaaS provider's automatic VM restarting feature for IaaS VMs
 - Mitigates: Apiserver VM shutdown or apiserver crashing
 - Mitigates: Supporting services VM shutdown or crashes
- Action: Use IaaS providers reliable storage (e.g. GCE PD or AWS EBS volume) for VMs with apiserver+etcd
 - Mitigates: Apiserver backing storage lost
- Action: Use [high-availability](#) configuration
 - Mitigates: Control plane node shutdown or control plane components (scheduler, API server, controller-manager) crashing

- Will tolerate one or more simultaneous node or component failures
- Mitigates: API server backing storage (i.e., etcd's data directory) lost
 - Assumes HA (highly-available) etcd configuration
- Action: Snapshot apiserver PDs/EBS-volumes periodically
 - Mitigates: Apiserver backing storage lost
 - Mitigates: Some cases of operator error
 - Mitigates: Some cases of Kubernetes software fault
- Action: use replication controller and services in front of pods
 - Mitigates: Node shutdown
 - Mitigates: Kubelet software fault
- Action: applications (containers) designed to tolerate unexpected restarts
 - Mitigates: Node shutdown
 - Mitigates: Kubelet software fault

What's next

- Learn about the metrics available in the [Resource Metrics Pipeline](#)
- Discover additional tools for [monitoring resource usage](#)
- Use Node Problem Detector to [monitor node health](#)
- Use `kubectl debug node` to [debug Kubernetes nodes](#)
- Use `cricctl` to [debug Kubernetes nodes](#)
- Get more information about [Kubernetes auditing](#)
- Use `telepresence` to [develop and debug services locally](#)

4.2.1 - Troubleshooting kubectl

This documentation is about investigating and diagnosing `kubectl` related issues. If you encounter issues accessing `kubectl` or connecting to your cluster, this document outlines various common scenarios and potential solutions to help identify and address the likely cause.

Before you begin

- You need to have a Kubernetes cluster.
- You also need to have `kubectl` installed - see [install tools](#)

Verify kubectl setup

Make sure you have installed and configured `kubectl` correctly on your local machine. Check the `kubectl` version to ensure it is up-to-date and compatible with your cluster.

Check `kubectl` version:

```
kubectl version
```

You'll see a similar output:

```
Client Version: version.Info{Major:"1", Minor:"27", GitVersion:"v1.27.4"
Kustomize Version: v5.0.1
Server Version: version.Info{Major:"1", Minor:"27", GitVersion:"v1.27.4"}
```

If you see `Unable to connect to the server: dial tcp <server-ip>:8443: i/o timeout`, instead of `Server Version`, you need to troubleshoot `kubectl` connectivity with your cluster.

Make sure you have installed the `kubectl` by following the [official documentation for installing kubectl](#), and you have properly configured the `$PATH` environment variable.

Check kubeconfig

The `kubectl` requires a `kubeconfig` file to connect to a Kubernetes cluster. The `kubeconfig` file is usually located under the `~/.kube/config` directory. Make sure that you have a valid `kubeconfig` file. If you don't have a `kubeconfig` file, you can obtain it from your Kubernetes administrator, or you can copy it from your Kubernetes control plane's `/etc/kubernetes/admin.conf` directory. If you have deployed your Kubernetes cluster on a cloud platform and lost your `kubeconfig` file, you can re-generate it using your cloud provider's tools. Refer the cloud provider's documentation for re-generating a `kubeconfig` file.

Check if the `$KUBECONFIG` environment variable is configured correctly. You can set `$KUBECONFIG` environment variable or use the `--kubeconfig` parameter with the `kubectl` to specify the directory of a `kubeconfig` file.

Check VPN connectivity

If you are using a Virtual Private Network (VPN) to access your Kubernetes cluster, make sure that your VPN connection is active and stable. Sometimes, VPN disconnections can lead to connection issues with the cluster. Reconnect to the VPN and try accessing the cluster again.

Authentication and authorization

If you are using the token based authentication and the kubectl is returning an error regarding the authentication token or authentication server address, validate the Kubernetes authentication token and the authentication server address are configured properly.

If kubectl is returning an error regarding the authorization, make sure that you are using the valid user credentials. And you have the permission to access the resource that you have requested.

Verify contexts

Kubernetes supports [multiple clusters and contexts](#). Ensure that you are using the correct context to interact with your cluster.

List available contexts:

```
kubectl config get-contexts
```

Switch to the appropriate context:

```
kubectl config use-context <context-name>
```

API server and load balancer

The [kube-apiserver](#) server is the central component of a Kubernetes cluster. If the API server or the load balancer that runs in front of your API servers is not reachable or not responding, you won't be able to interact with the cluster.

Check the if the API server's host is reachable by using `ping` command. Check cluster's network connectivity and firewall. If your are using a cloud provider for deploying the cluster, check your cloud provider's health check status for the cluster's API server.

Verify the status of the load balancer (if used) to ensure it is healthy and forwarding traffic to the API server.

TLS problems

The Kubernetes API server only serves HTTPS requests by default. In that case TLS problems may occur due to various reasons, such as certificate expiry or chain of trust validity.

You can find the TLS certificate in the kubeconfig file, located in the `~/.kube/config` directory. The `certificate-authority` attribute contains the CA certificate and the `client-certificate` attribute contains the client

certificate.

Verify the expiry of these certificates:

```
openssl x509 -noout -dates -in $(kubectl config view --minify --output
```

output:

```
notBefore=Sep 2 08:34:12 2023 GMT  
notAfter=Aug 31 08:34:12 2033 GMT
```

```
openssl x509 -noout -dates -in $(kubectl config view --minify --output
```

output:

```
notBefore=Sep 2 08:34:12 2023 GMT  
notAfter=Sep 2 08:34:12 2026 GMT
```

Verify kubectl helpers

Some kubectl authentication helpers provide easy access to Kubernetes clusters. If you have used such helpers and are facing connectivity issues, ensure that the necessary configurations are still present.

Check kubectl configuration for authentication details:

```
kubectl config view
```

If you previously used a helper tool (for example, `kubectl-oidc-login`), ensure that it is still installed and configured correctly.

4.2.2 - Resource metrics pipeline

For Kubernetes, the *Metrics API* offers a basic set of metrics to support automatic scaling and similar use cases. This API makes information available about resource usage for node and pod, including metrics for CPU and memory. If you deploy the Metrics API into your cluster, clients of the Kubernetes API can then query for this information, and you can use Kubernetes' access control mechanisms to manage permissions to do so.

The [HorizontalPodAutoscaler](#) (HPA) and [VerticalPodAutoscaler](#) (VPA) use data from the metrics API to adjust workload replicas and resources to meet customer demand.

You can also view the resource metrics using the [kubectl top](#) command.

Note: The Metrics API, and the metrics pipeline that it enables, only offers the minimum CPU and memory metrics to enable automatic scaling using HPA and / or VPA. If you would like to provide a more complete set of metrics, you can complement the simpler Metrics API by deploying a second [metrics pipeline](#) that uses the *Custom Metrics API*.

Figure 1 illustrates the architecture of the resource metrics pipeline.

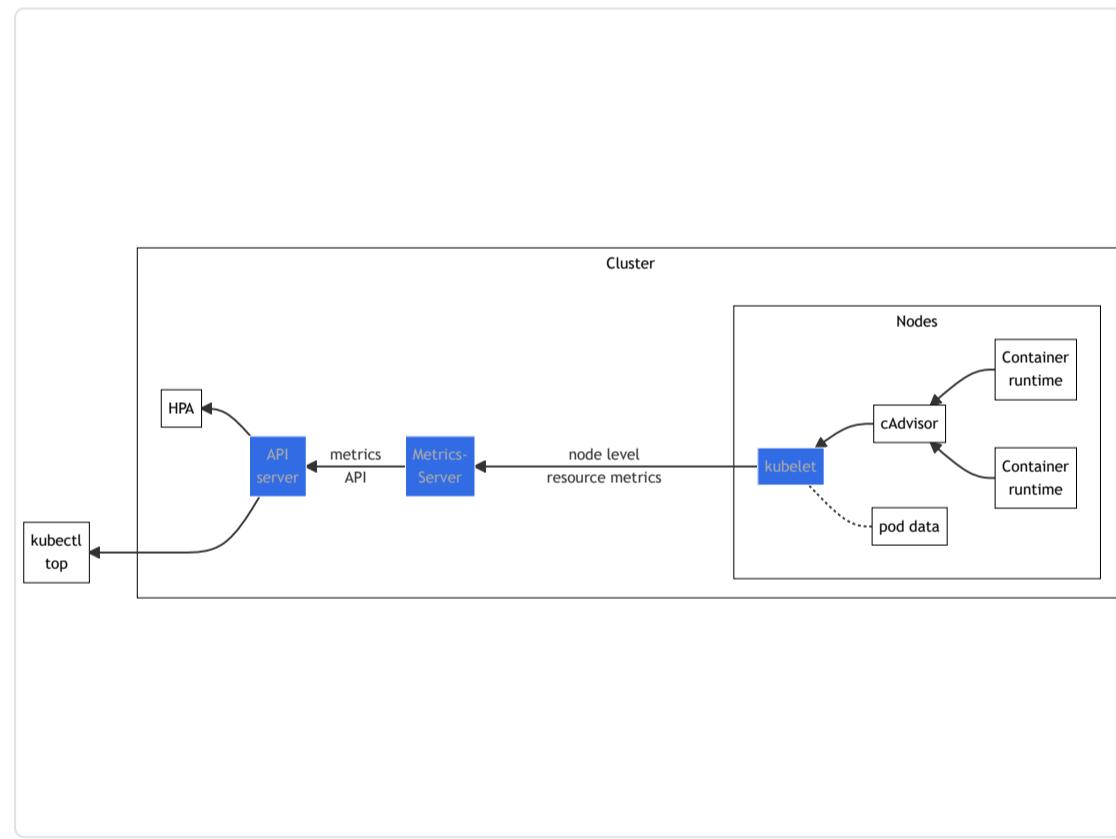


Figure 1. Resource Metrics Pipeline

The architecture components, from right to left in the figure, consist of the following:

- [cAdvisor](#): Daemon for collecting, aggregating and exposing container metrics included in Kubelet.
- [kubelet](#): Node agent for managing container resources. Resource metrics are accessible using the `/metrics/resource` and `/stats` kubelet API endpoints.
- [node level resource metrics](#): API provided by the kubelet for discovering and retrieving per-node summarized stats available through the `/metrics/resource` endpoint.
- [metrics-server](#): Cluster addon component that collects and aggregates resource metrics pulled from each kubelet. The API server serves Metrics API for use by HPA, VPA, and by the `kubectl top`

- command. Metrics Server is a reference implementation of the Metrics API.
- [Metrics API](#): Kubernetes API supporting access to CPU and memory used for workload autoscaling. To make this work in your cluster, you need an API extension server that provides the Metrics API.

Note: cAdvisor supports reading metrics from cgroups, which works with typical container runtimes on Linux. If you use a container runtime that uses another resource isolation mechanism, for example virtualization, then that container runtime must support [CRI Container Metrics](#) in order for metrics to be available to the kubelet.

Metrics API

FEATURE STATE: [Kubernetes 1.8 \[beta\]](#)

The metrics-server implements the Metrics API. This API allows you to access CPU and memory usage for the nodes and pods in your cluster. Its primary role is to feed resource usage metrics to K8s autoscaler components.

Here is an example of the Metrics API request for a `minikube` node piped through `jq` for easier reading:

```
kubectl get --raw "/apis/metrics.k8s.io/v1beta1/nodes/minikube" | jq '
```

Here is the same API call using `curl`:

```
curl http://localhost:8080/apis/metrics.k8s.io/v1beta1/nodes/minikube
```

Sample response:

```
{  
  "kind": "NodeMetrics",  
  "apiVersion": "metrics.k8s.io/v1beta1",  
  "metadata": {  
    "name": "minikube",  
    "selfLink": "/apis/metrics.k8s.io/v1beta1/nodes/minikube",  
    "creationTimestamp": "2022-01-27T18:48:43Z"  
  },  
  "timestamp": "2022-01-27T18:48:33Z",  
  "window": "30s",  
  "usage": {  
    "cpu": "487558164n",  
    "memory": "732212Ki"  
  }  
}
```

Here is an example of the Metrics API request for a `kube-scheduler-minikube` pod contained in the `kube-system` namespace and piped through `jq` for easier reading:

```
kubectl get --raw "/apis/metrics.k8s.io/v1beta1/namespaces/kube-system,
```

Here is the same API call using `curl`:

```
curl http://localhost:8080/apis/metrics.k8s.io/v1beta1/namespaces/kube
```

Sample response:

```
{
  "kind": "PodMetrics",
  "apiVersion": "metrics.k8s.io/v1beta1",
  "metadata": {
    "name": "kube-scheduler-minikube",
    "namespace": "kube-system",
    "selfLink": "/apis/metrics.k8s.io/v1beta1/namespaces/kube-system/pods/kube-scheduler-minikube",
    "creationTimestamp": "2022-01-27T19:25:00Z"
  },
  "timestamp": "2022-01-27T19:24:31Z",
  "window": "30s",
  "containers": [
    {
      "name": "kube-scheduler",
      "usage": {
        "cpu": "9559630n",
        "memory": "22244Ki"
      }
    }
  ]
}
```

The Metrics API is defined in the [k8s.io/metrics](#) repository. You must enable the [API aggregation layer](#) and register an [APIService](#) for the `metrics.k8s.io` API.

To learn more about the Metrics API, see [resource metrics API design](#), the [metrics-server repository](#) and the [resource metrics API](#).

Note: You must deploy the metrics-server or alternative adapter that serves the Metrics API to be able to access it.

Measuring resource usage

CPU

CPU is reported as the average core usage measured in cpu units. One cpu, in Kubernetes, is equivalent to 1 vCPU/Core for cloud providers, and 1 hyper-thread on bare-metal Intel processors.

This value is derived by taking a rate over a cumulative CPU counter provided by the kernel (in both Linux and Windows kernels). The time window used to calculate CPU is shown under `window` field in Metrics API.

To learn more about how Kubernetes allocates and measures CPU resources, see [meaning of CPU](#).

Memory

Memory is reported as the working set, measured in bytes, at the instant the metric was collected.

In an ideal world, the "working set" is the amount of memory in-use that cannot be freed under memory pressure. However, calculation of the working set varies by host OS, and generally makes heavy use of heuristics to produce an estimate.

The Kubernetes model for a container's working set expects that the container runtime counts anonymous memory associated with the container in question. The working set metric typically also includes some cached (file-backed) memory, because the host OS cannot always reclaim pages.

To learn more about how Kubernetes allocates and measures memory resources, see [meaning of memory](#).

Metrics Server

The metrics-server fetches resource metrics from the kubelets and exposes them in the Kubernetes API server through the Metrics API for use by the HPA and VPA. You can also view these metrics using the `kubectl top` command.

The metrics-server uses the Kubernetes API to track nodes and pods in your cluster. The metrics-server queries each node over HTTP to fetch metrics. The metrics-server also builds an internal view of pod metadata, and keeps a cache of pod health. That cached pod health information is available via the extension API that the metrics-server makes available.

For example with an HPA query, the metrics-server needs to identify which pods fulfill the label selectors in the deployment.

The metrics-server calls the [kubelet](#) API to collect metrics from each node. Depending on the metrics-server version it uses:

- Metrics resource endpoint `/metrics/resource` in version v0.6.0+ or
- Summary API endpoint `/stats/summary` in older versions

What's next

To learn more about the metrics-server, see the [metrics-server repository](#).

You can also check out the following:

- [metrics-server design](#)
- [metrics-server FAQ](#)
- [metrics-server known issues](#)
- [metrics-server releases](#)
- [Horizontal Pod Autoscaling](#)

To learn about how the kubelet serves node metrics, and how you can access those via the Kubernetes API, read [Node Metrics Data](#).

4.2.3 - Tools for Monitoring Resources

To scale an application and provide a reliable service, you need to understand how the application behaves when it is deployed. You can examine application performance in a Kubernetes cluster by examining the containers, [pods](#), [services](#), and the characteristics of the overall cluster. Kubernetes provides detailed information about an application's resource usage at each of these levels. This information allows you to evaluate your application's performance and where bottlenecks can be removed to improve overall performance.

In Kubernetes, application monitoring does not depend on a single monitoring solution. On new clusters, you can use [resource metrics](#) or [full metrics](#) pipelines to collect monitoring statistics.

Resource metrics pipeline

The resource metrics pipeline provides a limited set of metrics related to cluster components such as the [Horizontal Pod Autoscaler](#) controller, as well as the `kubectl top` utility. These metrics are collected by the lightweight, short-term, in-memory [metrics-server](#) and are exposed via the `metrics.k8s.io` API.

metrics-server discovers all nodes on the cluster and queries each node's [kubelet](#) for CPU and memory usage. The kubelet acts as a bridge between the Kubernetes master and the nodes, managing the pods and containers running on a machine. The kubelet translates each pod into its constituent containers and fetches individual container usage statistics from the container runtime through the container runtime interface. If you use a container runtime that uses Linux cgroups and namespaces to implement containers, and the container runtime does not publish usage statistics, then the kubelet can look up those statistics directly (using code from [cAdvisor](#)). No matter how those statistics arrive, the kubelet then exposes the aggregated pod resource usage statistics through the metrics-server Resource Metrics API. This API is served at `/metrics/resource/v1beta1` on the kubelet's authenticated and read-only ports.

Full metrics pipeline

A full metrics pipeline gives you access to richer metrics. Kubernetes can respond to these metrics by automatically scaling or adapting the cluster based on its current state, using mechanisms such as the Horizontal Pod Autoscaler. The monitoring pipeline fetches metrics from the kubelet and then exposes them to Kubernetes via an adapter by implementing either the `custom.metrics.k8s.io` or `external.metrics.k8s.io` API.

Kubernetes is designed to work with [OpenMetrics](#), which is one of the [CNCF Observability and Analysis - Monitoring Projects](#), built upon and carefully extending [Prometheus exposition format](#) in almost 100% backwards-compatible ways.

If you glance over at the [CNCF Landscape](#), you can see a number of monitoring projects that can work with Kubernetes by *scraping* metric data and using that to help you observe your cluster. It is up to you to select the tool or tools that suit your needs. The CNCF landscape for observability and analytics includes a mix of open-source software, paid-for software-as-a-service, and other commercial products.

When you design and implement a full metrics pipeline you can make that monitoring data available back to Kubernetes. For example, a HorizontalPodAutoscaler can use the processed metrics to work out how many Pods to run for a component of your workload.

Integration of a full metrics pipeline into your Kubernetes implementation is outside the scope of Kubernetes documentation because of the very wide scope of possible solutions.

The choice of monitoring platform depends heavily on your needs, budget, and technical resources. Kubernetes does not recommend any specific metrics pipeline; [many options](#) are available. Your monitoring system should be capable of handling the [OpenMetrics](#) metrics transmission standard, and needs to chosen to best fit in to your overall design and deployment of your infrastructure platform.

What's next

Learn about additional debugging tools, including:

- [Logging](#)
- [Monitoring](#)
- [Getting into containers via exec](#)
- [Connecting to containers via proxies](#)
- [Connecting to containers via port forwarding](#)
- [Inspect Kubernetes node with crictl](#)

4.2.4 - Monitor Node Health

Node Problem Detector is a daemon for monitoring and reporting about a node's health. You can run Node Problem Detector as a `DaemonSet` or as a standalone daemon. Node Problem Detector collects information about node problems from various daemons and reports these conditions to the API server as Node [Conditions](#) or as [Events](#).

To learn how to install and use Node Problem Detector, see [Node Problem Detector project documentation](#).

Before you begin

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

- [Killercode](#)
- [Play with Kubernetes](#)

Limitations

- Node Problem Detector uses the kernel log format for reporting kernel issues. To learn how to extend the kernel log format, see [Add support for another log format](#).

Enabling Node Problem Detector

Some cloud providers enable Node Problem Detector as an [Addon](#). You can also enable Node Problem Detector with `kubectl` or by creating an Addon `DaemonSet`.

Using `kubectl` to enable Node Problem Detector

`kubectl` provides the most flexible management of Node Problem Detector. You can overwrite the default configuration to fit it into your environment or to detect customized node problems. For example:

1. Create a Node Problem Detector configuration similar to `node-problem-detector.yaml` :

[debug/node-problem-detector.yaml](#) 

```

apiVersion: apps/v1
kind: DaemonSet
metadata:
  name: node-problem-detector-v0.1
  namespace: kube-system
  labels:
    k8s-app: node-problem-detector
    version: v0.1
    kubernetes.io/cluster-service: "true"
spec:
  selector:
    matchLabels:
      k8s-app: node-problem-detector
      version: v0.1
      kubernetes.io/cluster-service: "true"
  template:
    metadata:
      labels:
        k8s-app: node-problem-detector
        version: v0.1
        kubernetes.io/cluster-service: "true"
    spec:
      hostNetwork: true
      containers:
        - name: node-problem-detector
          image: registry.k8s.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/

```

Note: You should verify that the system log directory is right for your operating system distribution.

2. Start node problem detector with `kubectl`:

```
kubectl apply -f https://k8s.io/examples/debug/node-problem-detec
```

Using an Addon pod to enable Node Problem Detector

If you are using a custom cluster bootstrap solution and don't need to overwrite the default configuration, you can leverage the Addon pod to further automate the deployment.

Create `node-problem-detector.yaml`, and save the configuration in the Addon pod's directory `/etc/kubernetes/addons/node-problem-detector` on a control plane node.

Overwrite the configuration

The [default configuration](#) is embedded when building the Docker image of Node Problem Detector.

However, you can use a [ConfigMap](#) to overwrite the configuration:

1. Change the configuration files in `config/`
2. Create the `ConfigMap node-problem-detector-config`:

```
kubectl create configmap node-problem-detector-config --from-file
```

3. Change the `node-problem-detector.yaml` to use the `ConfigMap`:

```
debug/node-problem-detector-configmap.yaml ↗
```

```
apiVersion: apps/v1
kind: DaemonSet
metadata:
  name: node-problem-detector-v0.1
  namespace: kube-system
  labels:
    k8s-app: node-problem-detector
    version: v0.1
    kubernetes.io/cluster-service: "true"
spec:
  selector:
    matchLabels:
      k8s-app: node-problem-detector
      version: v0.1
      kubernetes.io/cluster-service: "true"
  template:
    metadata:
      labels:
        k8s-app: node-problem-detector
        version: v0.1
        kubernetes.io/cluster-service: "true"
    spec:
      hostNetwork: true
      containers:
        - name: node-problem-detector
          image: registry.k8s.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
              mountPath: /config
              readOnly: true
          volumes:
            - name: log
              hostPath:
                path: /var/log/
            - name: config # Define ConfigMap volume
              configMap:
                name: node-problem-detector-config
```

4. Recreate the Node Problem Detector with the new configuration file:

```
# If you have a node-problem-detector running, delete before recreating
kubectl delete -f https://k8s.io/examples/debug/node-problem-detector-v0.1.yaml
kubectl apply -f https://k8s.io/examples/debug/node-problem-detector-v0.1.yaml
```

Note: This approach only applies to a Node Problem Detector started with `kubectl`.

Overwriting a configuration is not supported if a Node Problem Detector runs as a cluster Addon. The Addon manager does not support `ConfigMap`.

Problem Daemons

A problem daemon is a sub-daemon of the Node Problem Detector. It monitors specific kinds of node problems and reports them to the Node Problem Detector. There are several types of supported problem daemons.

- A `SystemLogMonitor` type of daemon monitors the system logs and reports problems and metrics according to predefined rules. You can customize the configurations for different log sources such as [filelog](#), [kmsg](#), [kernel](#), [abrt](#), and [systemd](#).
- A `SystemStatsMonitor` type of daemon collects various health-related system stats as metrics. You can customize its behavior by updating its [configuration file](#).
- A `CustomPluginMonitor` type of daemon invokes and checks various node problems by running user-defined scripts. You can use different custom plugin monitors to monitor different problems and customize the daemon behavior by updating the [configuration file](#).
- A `HealthChecker` type of daemon checks the health of the kubelet and container runtime on a node.

Adding support for other log format

The system log monitor currently supports file-based logs, journald, and kmsg. Additional sources can be added by implementing a new [log watcher](#).

Adding custom plugin monitors

You can extend the Node Problem Detector to execute any monitor scripts written in any language by developing a custom plugin. The monitor scripts must conform to the plugin protocol in exit code and standard output. For more information, please refer to the [plugin interface proposal](#).

Exporter

An exporter reports the node problems and/or metrics to certain backends. The following exporters are supported:

- **Kubernetes exporter:** this exporter reports node problems to the Kubernetes API server. Temporary problems are reported as Events and permanent problems are reported as Node Conditions.
- **Prometheus exporter:** this exporter reports node problems and metrics locally as Prometheus (or OpenMetrics) metrics. You can specify the IP address and port for the exporter using command line arguments.
- **Stackdriver exporter:** this exporter reports node problems and metrics to the Stackdriver Monitoring API. The exporting behavior can be customized using a [configuration file](#).

Recommendations and restrictions

It is recommended to run the Node Problem Detector in your cluster to monitor node health. When running the Node Problem Detector, you can expect extra resource overhead on each node. Usually this is fine, because:

- The kernel log grows relatively slowly.
- A resource limit is set for the Node Problem Detector.
- Even under high load, the resource usage is acceptable. For more information, see the Node Problem Detector [benchmark result](#).

4.2.5 - Debugging Kubernetes nodes with crictl

FEATURE STATE: [Kubernetes v1.11 \[stable\]](#)

`crictl` is a command-line interface for CRI-compatible container runtimes. You can use it to inspect and debug container runtimes and applications on a Kubernetes node. `crictl` and its source are hosted in the [cri-tools](#) repository.

Before you begin

`crictl` requires a Linux operating system with a CRI runtime.

Installing crictl

You can download a compressed archive `crictl` from the cri-tools [release page](#), for several different architectures. Download the version that corresponds to your version of Kubernetes. Extract it and move it to a location on your system path, such as `/usr/local/bin/`.

General usage

The `crictl` command has several subcommands and runtime flags. Use `crictl help` or `crictl <subcommand> help` for more details.

You can set the endpoint for `crictl` by doing one of the following:

- Set the `--runtime-endpoint` and `--image-endpoint` flags.
- Set the `CONTAINER_RUNTIME_ENDPOINT` and `IMAGE_SERVICE_ENDPOINT` environment variables.
- Set the endpoint in the configuration file `/etc/crictl.yaml`. To specify a different file, use the `--config=PATH_TO_FILE` flag when you run `crictl`.

Note: If you don't set an endpoint, `crictl` attempts to connect to a list of known endpoints, which might result in an impact to performance.

You can also specify timeout values when connecting to the server and enable or disable debugging, by specifying `timeout` or `debug` values in the configuration file or using the `--timeout` and `--debug` command-line flags.

To view or edit the current configuration, view or edit the contents of `/etc/crictl.yaml`. For example, the configuration when using the `containerd` container runtime would be similar to this:

```
runtime-endpoint: unix:///var/run/containerd/containerd.sock
image-endpoint: unix:///var/run/containerd/containerd.sock
timeout: 10
debug: true
```

To learn more about `crictl`, refer to the [crictl documentation](#).

Example crictl commands

The following examples show some `crictl` commands and example output.

Warning: If you use `crictl` to create pod sandboxes or containers on a running Kubernetes cluster, the Kubelet will eventually delete them. `crictl` is not a general purpose workflow tool, but a tool that is useful for debugging.

List pods

List all pods:

```
crictl pods
```

The output is similar to this:

POD ID	CREATED	STATE	NAME
926f1b5a1d33a	About a minute ago	Ready	sh-84d7dcf
4dcc216c4adb	About a minute ago	Ready	nginx-6589
a86316e96fa89	17 hours ago	Ready	kube-proxy
919630b8f81f1	17 hours ago	Ready	nvidia-dev

List pods by name:

```
crictl pods --name nginx-65899c769f-wv2gp
```

The output is similar to this:

POD ID	CREATED	STATE	NAME
4dcc216c4adb	2 minutes ago	Ready	nginx-6589

List pods by label:

```
crictl pods --label run=nginx
```

The output is similar to this:

POD ID	CREATED	STATE	NAME
4dcc216c4adb	2 minutes ago	Ready	nginx-6589

List images

List all images:

```
crictl images
```

The output is similar to this:

IMAGE	TAG	IMAGE ID
busybox	latest	8c811b4ae
k8s-gcrio.azureedge.net/hyperkube-amd64	v1.10.3	e179bbfe5
k8s-gcrio.azureedge.net/pause-amd64	3.1	da86e6ba6
nginx	latest	cd5239a09

List images by repository:

```
crictl images nginx
```

The output is similar to this:

IMAGE	TAG	IMAGE ID	SIZE
nginx	latest	cd5239a0906a6	109MB

Only list image IDs:

```
crictl images -q
```

The output is similar to this:

```
sha256:8c811b4aec35f259572d0f79207bc0678df4c736eeec50bc9fec37ed936a472a
sha256:e179bbfe5d238de6069f3b03fccbecc3fb4f2019af741bfff1233c4d7b2970c5
sha256:da86e6ba6ca197bf6bc5e9d900feb906b133eaa4750e6bed647b0fbe50ed43e
sha256:cd5239a0906a6ccf0562354852fae04bc5b52d72a2aff9a871ddb6bd57553569
```

List containers

List all containers:

```
crictl ps -a
```

The output is similar to this:

CONTAINER ID	IMAGE
1f73f2d81bf98	busybox@sha256:141c253bc4c3fd0a201d32dc1f493bcf3ff1
9c5951df22c78	busybox@sha256:141c253bc4c3fd0a201d32dc1f493bcf3ff1
87d3992f84f74	nginx@sha256:d0a8828cccb73397acb0073bf34f4d7d8aa315
1941fb4da154f	k8s-gcrio.azureedge.net/hyperkube-amd64@sha256:00d8

List running containers:

```
crictl ps
```

The output is similar to this:

CONTAINER ID	IMAGE
1f73f2d81bf98	busybox@sha256:141c253bc4c3fd0a201d32dc1f493bcf3ff1
87d3992f84f74	nginx@sha256:d0a8828ccb73397acb0073bf34f4d7d8aa315
1941fb4da154f	k8s-gcrio.azureedge.net/hyperkube-amd64@sha256:00d8

Execute a command in a running container

```
crictl exec -i -t 1f73f2d81bf98 ls
```

The output is similar to this:

```
bin dev etc home proc root sys tmp usr var
```

Get a container's logs

Get all container logs:

```
crictl logs 87d3992f84f74
```

The output is similar to this:

```
10.240.0.96 - - [06/Jun/2018:02:45:49 +0000] "GET / HTTP/1.1" 200 612 '
10.240.0.96 - - [06/Jun/2018:02:45:50 +0000] "GET / HTTP/1.1" 200 612 '
10.240.0.96 - - [06/Jun/2018:02:45:51 +0000] "GET / HTTP/1.1" 200 612 '
```

Get only the latest `N` lines of logs:

```
crictl logs --tail=1 87d3992f84f74
```

The output is similar to this:

```
10.240.0.96 - - [06/Jun/2018:02:45:51 +0000] "GET / HTTP/1.1" 200 612 '
```

Run a pod sandbox

Using `crictl` to run a pod sandbox is useful for debugging container runtimes. On a running Kubernetes cluster, the sandbox will eventually be stopped and deleted by the Kubelet.

1. Create a JSON file like the following:

```
{  
  "metadata": {  
    "name": "nginx-sandbox",  
    "namespace": "default",  
    "attempt": 1,  
    "uid": "hdishd83djaidwnduwk28bcsb"  
  },  
  "log_directory": "/tmp",  
  "linux": {}  
}
```

2. Use the `crictl runp` command to apply the JSON and run the sandbox.

```
crictl runp pod-config.json
```

The ID of the sandbox is returned.

Create a container

Using `crictl` to create a container is useful for debugging container runtimes. On a running Kubernetes cluster, the sandbox will eventually be stopped and deleted by the Kubelet.

1. Pull a busybox image

```
crictl pull busybox
```

```
Image is up to date for busybox@sha256:141c253bc4c3fd0a201d32dc1f...
```

2. Create configs for the pod and the container:

Pod config:

```
{  
  "metadata": {  
    "name": "busybox-sandbox",  
    "namespace": "default",  
    "attempt": 1,  
    "uid": "aewi4aeThua7ooShohbo1phoj"  
  },  
  "log_directory": "/tmp",  
  "linux": {}  
}
```

Container config:

```
{
  "metadata": {
    "name": "busybox"
  },
  "image": {
    "image": "busybox"
  },
  "command": [
    "top"
  ],
  "log_path": "busybox.log",
  "linux": {
  }
}
```

3. Create the container, passing the ID of the previously-created pod, the container config file, and the pod config file. The ID of the container is returned.

```
crictl create f84dd361f8dc51518ed291fbadd6db537b0496536c1d2d6c05f
```

4. List all containers and verify that the newly-created container has its state set to `Created`.

```
crictl ps -a
```

The output is similar to this:

CONTAINER ID	IMAGE	CREATED	STATE
3e025dd50a72d	busybox	32 seconds ago	Created

Start a container

To start a container, pass its ID to `crictl start`:

```
crictl start 3e025dd50a72d956c4f14881fbb5b1080c9275674e95fb67f965f6478a957d60
```

The output is similar to this:

```
3e025dd50a72d956c4f14881fbb5b1080c9275674e95fb67f965f6478a957d60
```

Check the container has its state set to `Running`.

```
crictl ps
```

The output is similar to this:

CONTAINER ID	IMAGE	CREATED	STATE	NAME	ATTEMPT
3e025dd50a72d	busybox	About a minute ago	Running	busybox	0

What's next

- [Learn more about `cricctl`.](#)
- [Map `docker` CLI commands to `cricctl`.](#)

4.2.6 - Auditing

Kubernetes *auditing* provides a security-relevant, chronological set of records documenting the sequence of actions in a cluster. The cluster audits the activities generated by users, by applications that use the Kubernetes API, and by the control plane itself.

Auditing allows cluster administrators 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 records begin their lifecycle inside the [kube-apiserver](#) component. Each request on each stage of its execution generates an audit 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 defined 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 configuration of an [Audit Event configuration](#) is different from the [Event API object](#).

The audit logging feature increases the memory consumption of the API server because some context required for auditing is stored for each request. 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 defined 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/audit-policy.yaml](#) 

```
apiVersion: audit.k8s.io/v1 # This is required.
kind: Policy
# Don't generate audit events for all requests in RequestReceived stage
omitStages:
- "RequestReceived"
rules:
# Log pod changes at RequestResponse Level
- level: RequestResponse
resources:
- group: ""
  # Resource "pods" doesn't match requests to any subresource of pods
  # which is consistent with the RBAC policy.
  resources: ["pods"]
# Log "pods/Log", "pods/status" at Metadata Level
- level: Metadata
resources:
- group: ""
  resources: ["pods/log", "pods/status"]

# Don't log requests to a configmap called "controller-leader"
- level: None
resources:
- group: ""
  resources: ["configmaps"]
  resourceNames: ["controller-leader"]

# Don't log watch requests by the "system:kube-proxy" on endpoints or services
- level: None
users: ["system:kube-proxy"]
verbs: ["watch"]
resources:
- group: "" # core API group
  resources: ["endpoints", "services"]

# Don't log authenticated requests to certain non-resource URL paths
- level: None
userGroups: ["system:authenticated"]
nonResourceURLs:
- "/api*" # Wildcard matching.
- "/version"

# Log the request body of configmap changes in kube-system.
- level: Request
resources:
- group: "" # core API group
  resources: ["configmaps"]
# This rule only applies to resources in the "kube-system" namespace
# The empty string "" can be used to select non-namespaced resources
namespaces: ["kube-system"]

# Log configmap and secret changes in all other namespaces at the Metadata level
- level: Metadata
resources:
- group: "" # core API group
  resources: ["secrets", "configmaps"]

# Log all other resources in core and extensions at the Request Level
- level: Request
resources:
- group: "" # core API group
- group: "extensions" # Version of group should NOT be included.

# A catch-all rule to log all other requests at the Metadata Level.
- level: Metadata
# Long-running requests like watches that fall under this rule will
# generate an audit event in RequestReceived.
```

```
omitStages:  
- "RequestReceived"
```

You can use a minimal audit policy file to log all requests at the `Metadata` level:

```
# Log all requests at the Metadata Level.  
apiVersion: audit.k8s.io/v1  
kind: Policy  
rules:  
- level: Metadata
```

If you're crafting your own audit profile, you can use the audit profile for Google Container-Optimized OS as a starting point. You can check the [configure-helper.sh](#) script, which generates an audit policy file. You can see most of the audit policy file by looking directly at the script.

You can also refer to the [Policy configuration reference](#) for details about the fields defined.

Audit backends

Audit backends persist audit events to an external storage. Out of the box, the kube-apiserver provides two backends:

- Log backend, which writes events into the filesystem
- Webhook backend, which sends events to an external HTTP API

In all cases, audit events follow a structure defined by the Kubernetes API in the [audit.k8s.io API group](#).

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/terminationMessagePo  
 }  
]
```

Log backend

The log backend writes audit events to a file in [JSONlines](#) format. You can configure the log audit backend using the following `kube-apiserver` flags:

- `--audit-log-path` specifies the log file path that log backend uses to write audit events. Not specifying this flag disables log backend. `-` means standard out
- `--audit-log-maxage` defined the maximum number of days to retain old audit log files
- `--audit-log-maxbackup` defines the maximum number of audit log files to retain
- `--audit-log-maxsize` defines the maximum size in megabytes of the audit log file before it gets rotated

If your cluster's control plane runs the kube-apiserver as a Pod, remember to mount the `hostPath` to the location of the policy file and log file, so that audit records are persisted. For example:

```
- --audit-policy-file=/etc/kubernetes/audit-policy.yaml
- --audit-log-path=/var/log/kubernetes/audit/audit.log
```

then mount the volumes:

```
...
volumeMounts:
- mountPath: /etc/kubernetes/audit-policy.yaml
  name: audit
  readOnly: true
- mountPath: /var/log/kubernetes/audit/
  name: audit-log
  readOnly: false
```

and finally configure the `hostPath`:

```
...
volumes:
- name: audit
  hostPath:
    path: /etc/kubernetes/audit-policy.yaml
    type: File

- name: audit-log
  hostPath:
    path: /var/log/kubernetes/audit/
    type: DirectoryOrCreate
```

Webhook backend

The webhook audit backend sends audit events to a remote web API, which is assumed to be a form of the Kubernetes API, including means of authentication. You can configure a webhook audit backend using the following kube-apiserver flags:

- `--audit-webhook-config-file` specifies the path to a file with a webhook configuration. The webhook configuration is effectively a specialized [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.

Event batching

Both log and webhook backends support batching. Using webhook as an example, here's the list of available flags. To get the same flag for log backend, replace `webhook` with `log` in the flag name. By default, batching is enabled in `webhook` and disabled in `log`. Similarly, by default throttling is enabled in `webhook` and disabled in `log`.

- `--audit-webhook-mode` defines the buffering strategy. One of the following:
 - `batch` - buffer events and asynchronously process them in batches. This is the default.
 - `blocking` - block API server responses on processing each individual event.
 - `blocking-strict` - Same as blocking, but when there is a failure during audit logging at the `RequestReceived` stage, the whole request to the kube-apiserver fails.

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 API server.

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 queries per second. 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; that is: 10 batches, or 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.

Log entry truncation

Both log and webhook backends support limiting the size of events that are logged. As an example, the following is the list of flags available for the log backend:

- `audit-log-truncate-enabled` whether event and batch truncating is enabled.
- `audit-log-truncate-max-batch-size` maximum size in bytes of the batch sent to the underlying backend.
- `audit-log-truncate-max-event-size` maximum size in bytes of the audit event sent to the underlying backend.

By default truncate is disabled in both `webhook` and `log`, a cluster administrator should set `audit-log-truncate-enabled` or `audit-webhook-truncate-enabled` to enable the feature.

What's next

- Learn about [Mutating webhook auditing annotations](#).
- Learn more about [Event](#) and the [Policy](#) resource types by reading the Audit configuration reference.

4.2.7 - Debugging Kubernetes Nodes With Kubectl

This page shows how to debug a [node](#) running on the Kubernetes cluster using `kubectl debug` command.

Before you begin

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

- [Killercoda](#)
- [Play with Kubernetes](#)

Your Kubernetes server must be at or later than version 1.2. To check the version, enter `kubectl version`.

You need to have permission to create Pods and to assign those new Pods to arbitrary nodes. You also need to be authorized to create Pods that access filesystems from the host.

Debugging a Node using `kubectl debug node`

Use the `kubectl debug node` command to deploy a Pod to a Node that you want to troubleshoot. This command is helpful in scenarios where you can't access your Node by using an SSH connection. When the Pod is created, the Pod opens an interactive shell on the Node. To create an interactive shell on a Node named "mynode", run:

```
kubectl debug node/mynode -it --image=ubuntu
```

```
Creating debugging pod node-debugger-mynode-pdx84 with container debugger
If you don't see a command prompt, try pressing enter.
root@mynode:/#
```

The `debug` command helps to gather information and troubleshoot issues. Commands that you might use include `ip`, `ifconfig`, `nc`, `ping`, and `ps` and so on. You can also install other tools, such as `mtr`, `tcpdump`, and `curl`, from the respective package manager.

Note: The debug commands may differ based on the image the debugging pod is using and these commands might need to be installed.

The debugging Pod can access the root filesystem of the Node, mounted at `/host` in the Pod. If you run your kubelet in a filesystem namespace, the debugging Pod sees the root for that namespace, not for the entire node.

For a typical Linux node, you can look at the following paths to find relevant logs:

/host/var/log/kubelet.log

Logs from the `kubelet`, responsible for running containers on the node.

/host/var/log/kube-proxy.log

Logs from `kube-proxy`, which is responsible for directing traffic to Service endpoints.

/host/var/log/containerd.log

Logs from the `containerd` process running on the node.

/host/var/log/syslog

Shows general messages and information regarding the system.

/host/var/log/kern.log

Shows kernel logs.

When creating a debugging session on a Node, keep in mind that:

- `kubectl debug` automatically generates the name of the new pod, based on the name of the node.
- The root filesystem of the Node will be mounted at `/host`.
- Although the container runs in the host IPC, Network, and PID namespaces, the pod isn't privileged. This means that reading some process information might fail because access to that information is restricted to superusers. For example, `chroot /host` will fail. If you need a privileged pod, create it manually.

Cleaning up

When you finish using the debugging Pod, delete it:

```
kubectl get pods
```

NAME	READY	STATUS	RESTARTS	AGE
node-debugger-mynode-pdx84	0/1	Completed	0	8m1s

```
# Change the pod name accordingly  
kubectl delete pod node-debugger-mynode-pdx84 --now
```

```
pod "node-debugger-mynode-pdx84" deleted
```

Note: The `kubectl debug node` command won't work if the Node is down (disconnected from the network, or kubelet dies and won't restart, etc.). Check [debugging a down/unreachable node](#) in that case.

4.2.8 - Developing and debugging services locally using telepresence

Note: This section links to third party projects that provide functionality required by Kubernetes. The Kubernetes project authors aren't responsible for these projects, which are listed alphabetically. To add a project to this list, read the [content guide](#) before submitting a change. [More information.](#)

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](#) in order to run debugging tools.

`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

- Kubernetes cluster is installed
- `kubectl` is configured to communicate with the cluster
- [Telepresence](#) is installed

Connecting your local machine to a remote Kubernetes cluster

After installing `telepresence`, run `telepresence connect` to launch its Daemon and connect your local workstation to the cluster.

```
$ telepresence connect  
Launching Telepresence Daemon  
...  
Connected to context default (https://<cluster public IP>)
```

You can curl services using the Kubernetes syntax e.g. `curl -ik https://kubernetes.default`

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 `telepresence intercept $SERVICE_NAME --port $LOCAL_PORT:$REMOTE_PORT` command to create an "intercept" for rerouting remote service traffic.

Where:

- `$SERVICE_NAME` is the name of your local service
- `$LOCAL_PORT` is the port that your service is running on your local workstation
- And `$REMOTE_PORT` is the port your service listens to in the cluster

Running this command tells Telepresence to send remote traffic to your local service instead of the service in the remote Kubernetes cluster. Make edits to your service source code locally, save, and see the corresponding changes when accessing your remote application take effect immediately. You can also run your local service using a debugger or any other local development tool.

How does Telepresence work?

Telepresence installs a traffic-agent sidecar next to your existing application's container running in the remote cluster. It then captures all traffic requests going into the Pod, and instead of forwarding this to the application in the remote cluster, it routes all traffic (when you create a [global intercept](#) or a subset of the traffic (when you create a [personal intercept](#)) to your local development environment.

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.

For further reading, visit the [Telepresence website](#).

4.2.9 - Windows debugging tips

Node-level troubleshooting

1. My Pods are stuck at "Container Creating" or restarting over and over

Ensure that your pause image is compatible with your Windows OS version. See [Pause container](#) to see the latest / recommended pause image and/or get more information.

Note: If using containerd as your container runtime the pause image is specified in the `plugins.plugins.cri.sandbox_image` field of the `config.toml` configuration file.

2. My pods show status as `ErrImgPull` or `ImagePullBackOff`

Ensure that your Pod is getting scheduled to a [compatible](#) Windows Node.

More information on how to specify a compatible node for your Pod can be found in [this guide](#).

Network troubleshooting

1. My Windows Pods do not have network connectivity

If you are using virtual machines, ensure that MAC spoofing is **enabled** on all the VM network adapter(s).

2. My Windows Pods cannot ping external resources

Windows Pods do not have outbound rules programmed for the ICMP protocol. However, TCP/UDP is supported. When trying to demonstrate connectivity to resources outside of the cluster, substitute `ping <IP>` with corresponding `curl <IP>` commands.

If you are still facing problems, most likely your network configuration in [cni.conf](#) deserves some extra attention. You can always edit this static file. The configuration update will apply to any new Kubernetes resources.

One of the Kubernetes networking requirements (see [Kubernetes model](#)) is for cluster communication to occur without NAT internally.

To honor this requirement, there is an [ExceptionList](#) for all the communication where you do not want outbound NAT to occur.

However, this also means that you need to exclude the external IP you are trying to query from the `ExceptionList`. Only then will the traffic originating from your Windows pods be SNAT'ed correctly to receive a response from the outside world. In this regard, your `ExceptionList` in `cni.conf` should look as follows:

```
"ExceptionList": [
    "10.244.0.0/16",  # Cluster subnet
    "10.96.0.0/12",   # Service subnet
    "10.127.130.0/24" # Management (host) subnet
]
```

3. My Windows node cannot access `NodePort` type Services

Local NodePort access from the node itself fails. This is a known limitation. NodePort access works from other nodes or external clients.

4. vNICs and HNS endpoints of containers are being deleted

This issue can be caused when the `hostname-override` parameter is not passed to [kube-proxy](#). To resolve it, users need to pass the hostname to kube-proxy as follows:

```
C:\k\kube-proxy.exe --hostname-override=$(hostname)
```

5. My Windows node cannot access my services using the service IP

This is a known limitation of the networking stack on Windows. However, Windows Pods can access the Service IP.

6. No network adapter is found when starting the kubelet

The Windows networking stack needs a virtual adapter for Kubernetes networking to work. If the following commands return no results (in an admin shell), virtual network creation — a necessary prerequisite for the kubelet to work — has failed:

```
Get-HnsNetwork | ? Name -ieq "cbr0"  
Get-NetAdapter | ? Name -Like "vEthernet (Ethernet*)"
```

Often it is worthwhile to modify the [InterfaceName](#) parameter of the `start.ps1` script, in cases where the host's network adapter isn't "Ethernet". Otherwise, consult the output of the `start-kubelet.ps1` script to see if there are errors during virtual network creation.

7. DNS resolution is not properly working

Check the DNS limitations for Windows in [this section](#).

8. `kubectl port-forward` fails with "unable to do port forwarding: wincat not found"

This was implemented in Kubernetes 1.15 by including `wincat.exe` in the pause infrastructure container `mcr.microsoft.com/oss/kubernetes/pause:3.6`. Be sure to use a supported version of Kubernetes. If you would like to build your own pause infrastructure container be sure to include [wincat](#).

9. My Kubernetes installation is failing because my Windows Server node is behind a proxy

If you are behind a proxy, the following PowerShell environment variables must be defined:

```
[Environment]::SetEnvironmentVariable("HTTP_PROXY", "http://proxy  
[Environment]::SetEnvironmentVariable("HTTPS_PROXY", "http://prox
```

Flannel troubleshooting

1. With Flannel, my nodes are having issues after rejoining a cluster

Whenever a previously deleted node is being re-joined to the cluster, flanneld tries to assign a new pod subnet to the node. Users should remove the old pod subnet configuration files in the following paths:

```
Remove-Item C:\k\SourceVip.json  
Remove-Item C:\k\SourceVipRequest.json
```

2. Flanneld is stuck in "Waiting for the Network to be created"

There are numerous reports of this [issue](#); most likely it is a timing issue for when the management IP of the flannel network is set. A workaround is to relaunch `start.ps1` or relaunch it manually as follows:

```
[Environment]::SetEnvironmentVariable("NODE_NAME", "<Windows_Work  
C:\flannel\flanneld.exe --kubeconfig-file=c:\k\config --iface=<Wi
```

3. My Windows Pods cannot launch because of missing

`/run/flannel/subnet.env`

This indicates that Flannel didn't launch correctly. You can either try to restart `flanneld.exe` or you can copy the files over manually from `/run/flannel/subnet.env` on the Kubernetes master to `C:\run\flannel\subnet.env` on the Windows worker node and modify the `FLANNEL_SUBNET` row to a different number. For example, if node subnet 10.244.4.1/24 is desired:

```
FLANNEL_NETWORK=10.244.0.0/16  
FLANNEL_SUBNET=10.244.4.1/24  
FLANNEL_MTU=1500  
FLANNEL_IPMASQ=true
```

Further investigation

If these steps don't resolve your problem, you can get help running Windows containers on Windows nodes in Kubernetes through:

- StackOverflow [Windows Server Container](#) topic
- Kubernetes Official Forum discuss.kubernetes.io
- Kubernetes Slack [#SIG-Windows Channel](#)

5 - Manage Kubernetes Objects

Declarative and imperative paradigms for interacting with the Kubernetes API.

5.1 - Declarative Management of Kubernetes Objects Using Configuration Files

Kubernetes objects can be created, updated, and deleted by storing multiple object configuration files in a directory and using `kubectl apply` to recursively create and update those objects as needed. This method retains writes made to live objects without merging the changes back into the object configuration files. `kubectl diff` also gives you a preview of what changes `apply` will make.

Before you begin

Install [kubectl](#).

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

- [KillerCoda](#)
- [Play with Kubernetes](#)

To check the version, enter `kubectl version`.

Trade-offs

The `kubectl` tool supports three kinds of object management:

- Imperative commands
- Imperative object configuration
- Declarative object configuration

See [Kubernetes Object Management](#) for a discussion of the advantages and disadvantage of each kind of object management.

Overview

Declarative object configuration requires a firm understanding of the Kubernetes object definitions and configuration. Read and complete the following documents if you have not already:

- [Managing Kubernetes Objects Using Imperative Commands](#)
- [Imperative Management of Kubernetes Objects Using Configuration Files](#)

Following are definitions for terms used in this document:

- *object configuration file / configuration file*: A file that defines the configuration for a Kubernetes object. This topic shows how to pass configuration files to `kubectl apply`. Configuration files are typically stored in source control, such as Git.
- *live object configuration / live configuration*: The live configuration values of an object, as observed by the Kubernetes cluster. These are kept in the Kubernetes cluster storage, typically etcd.
- *declarative configuration writer / declarative writer*: A person or software component that makes updates to a live object. The live writers referred to in this topic make changes to object configuration files and run `kubectl apply` to write the changes.

How to create objects

Use `kubectl apply` to create all objects, except those that already exist, defined by configuration files in a specified directory:

```
kubectl apply -f <directory>
```

This sets the `kubectl.kubernetes.io/last-applied-configuration: '{...}'` annotation on each object. The annotation contains the contents of the object configuration file that was used to create the object.

Note: Add the `-R` flag to recursively process directories.

Here's an example of an object configuration file:

[application/simple_deployment.yaml](#) 

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-deployment
spec:
  selector:
    matchLabels:
      app: nginx
  minReadySeconds: 5
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
        - name: nginx
          image: nginx:1.14.2
          ports:
            - containerPort: 80
```

Run `kubectl diff` to print the object that will be created:

```
kubectl diff -f https://k8s.io/examples/application/simple_deployment.]
```

Note:

`diff` uses [server-side dry-run](#), which needs to be enabled on `kube-apiserver`.

Since `diff` performs a server-side apply request in dry-run mode, it requires granting `PATCH`, `CREATE`, and `UPDATE` permissions. See [Dry-Run Authorization](#) for details.

Create the object using `kubectl apply`:

```
kubectl apply -f https://k8s.io/examples/application/simple_deployment
```

Print the live configuration using `kubectl get`:

```
kubectl get -f https://k8s.io/examples/application/simple_deployment.ya
```

The output shows that the `kubectl.kubernetes.io/last-applied-configuration` annotation was written to the live configuration, and it matches the configuration file:

```

kind: Deployment
metadata:
  annotations:
    # ...
    # This is the json representation of simple_deployment.yaml
    # It was written by kubectl apply when the object was created
kubectl.kubernetes.io/last-applied-configuration: |
  {"apiVersion": "apps/v1", "kind": "Deployment",
   "metadata": {"annotations": {}, "name": "nginx-deployment", "namespace": "default"}, "spec": {"minReadySeconds": 5, "selector": {"matchLabels": {"app": "nginx"}}, "template": {"spec": {"containers": [{"image": "nginx:1.14.2", "name": "nginx", "ports": [{"containerPort": 80}]}]}}}}
  # ...
spec:
  # ...
minReadySeconds: 5
selector:
  matchLabels:
    # ...
    app: nginx
template:
  metadata:
    # ...
    labels:
      app: nginx
  spec:
    containers:
    - image: nginx:1.14.2
      # ...
      name: nginx
    ports:
    - containerPort: 80
      # ...
      # ...
    # ...
  # ...
# ...

```

How to update objects

You can also use `kubectl apply` to update all objects defined in a directory, even if those objects already exist. This approach accomplishes the following:

1. Sets fields that appear in the configuration file in the live configuration.
2. Clears fields removed from the configuration file in the live configuration.

```

kubectl diff -f <directory>
kubectl apply -f <directory>

```

Note: Add the `-R` flag to recursively process directories.

Here's an example configuration file:

[application/simple_deployment.yaml](#) 

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-deployment
spec:
  selector:
    matchLabels:
      app: nginx
  minReadySeconds: 5
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
        - name: nginx
          image: nginx:1.14.2
          ports:
            - containerPort: 80
```

Create the object using `kubectl apply` :

```
kubectl apply -f https://k8s.io/examples/application/simple_deployment
```

Note: For purposes of illustration, the preceding command refers to a single configuration file instead of a directory.

Print the live configuration using `kubectl get` :

```
kubectl get -f https://k8s.io/examples/application/simple_deployment.y
```

The output shows that the `kubectl.kubernetes.io/last-applied-configuration` annotation was written to the live configuration, and it matches the configuration file:

```
kind: Deployment
metadata:
  annotations:
    # ...
    # This is the json representation of simple_deployment.yaml
    # It was written by kubectl apply when the object was created
  kubectl.kubernetes.io/last-applied-configuration: |
    {"apiVersion": "apps/v1", "kind": "Deployment",
     "metadata": {"annotations": {}, "name": "nginx-deployment", "namespace": "default"}, "spec": {"minReadySeconds": 5, "selector": {"matchLabels": {"app": "nginx"}}, "template": {"spec": {"containers": [{"image": "nginx:1.14.2", "name": "nginx", "ports": [{"containerPort": 80}]}]}}}}
    # ...
  spec:
    # ...
  minReadySeconds: 5
  selector:
    matchLabels:
      # ...
      app: nginx
  template:
    metadata:
      # ...
      labels:
        app: nginx
    spec:
      containers:
        - image: nginx:1.14.2
          # ...
          name: nginx
        ports:
          - containerPort: 80
            # ...
        # ...
      # ...
    # ...
  # ...
```

Directly update the `replicas` field in the live configuration by using `kubectl scale`. This does not use `kubectl apply`:

```
kubectl scale deployment/nginx-deployment --replicas=2
```

Print the live configuration using `kubectl get`:

```
kubectl get deployment nginx-deployment -o yaml
```

The output shows that the `replicas` field has been set to 2, and the `last-applied-configuration` annotation does not contain a `replicas` field:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  annotations:
    # ...
    # note that the annotation does not contain replicas
    # because it was not updated through apply
  kubectl.kubernetes.io/last-applied-configuration: |
    {"apiVersion": "apps/v1", "kind": "Deployment",
     "metadata": {"annotations": {}, "name": "nginx-deployment", "namespace": "default"}, "spec": {"minReadySeconds": 5, "selector": {"matchLabels": {"app": "nginx"}}, "template": {"spec": {"containers": [{"image": "nginx:1.14.2", "name": "nginx", "ports": [{"containerPort": 80}]}]}}}
    # ...
  spec:
    replicas: 2 # written by scale
    # ...
    minReadySeconds: 5
    selector:
      matchLabels:
        # ...
        app: nginx
    template:
      metadata:
        # ...
        labels:
          app: nginx
      spec:
        containers:
          - image: nginx:1.14.2
            # ...
            name: nginx
        ports:
          - containerPort: 80
    # ...
```

Update the `simple_deployment.yaml` configuration file to change the image from `nginx:1.14.2` to `nginx:1.16.1`, and delete the `minReadySeconds` field:

[application/update_deployment.yaml](#) 

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-deployment
spec:
  selector:
    matchLabels:
      app: nginx
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
        - name: nginx
          image: nginx:1.16.1 # update the image
        ports:
          - containerPort: 80
```

Apply the changes made to the configuration file:

```
kubectl diff -f https://k8s.io/examples/application/update_deployment.yaml  
kubectl apply -f https://k8s.io/examples/application/update_deployment.yaml
```

Print the live configuration using `kubectl get`:

```
kubectl get -f https://k8s.io/examples/application/update_deployment.yaml
```

The output shows the following changes to the live configuration:

- The `replicas` field retains the value of 2 set by `kubectl scale`. This is possible because it is omitted from the configuration file.
- The `image` field has been updated to `nginx:1.16.1` from `nginx:1.14.2`.
- The `last-applied-configuration` annotation has been updated with the new image.
- The `minReadySeconds` field has been cleared.
- The `last-applied-configuration` annotation no longer contains the `minReadySeconds` field.

```
apiVersion: apps/v1
kind: Deployment
metadata:
  annotations:
    # ...
    # The annotation contains the updated image to nginx 1.16.1,
    # but does not contain the updated replicas to 2
  kubectl.kubernetes.io/last-applied-configuration: |
    {"apiVersion": "apps/v1", "kind": "Deployment",
     "metadata": {"annotations": {}, "name": "nginx-deployment", "namespace": "default"}, "spec": {"selector": {"matchLabels": {"app": "nginx"}}, "template": {"metadata": {"labels": {"app": "nginx"}}, "spec": {"containers": [{"image": "nginx:1.16.1", "name": "nginx", "ports": [{"containerPort": 80}]}]}}}}
    # ...
spec:
  replicas: 2 # Set by `kubectl scale`. Ignored by `kubectl apply`.
  # minReadySeconds cleared by `kubectl apply`
  # ...
  selector:
    matchLabels:
      # ...
      app: nginx
  template:
    metadata:
      # ...
      labels:
        app: nginx
    spec:
      containers:
        - image: nginx:1.16.1 # Set by `kubectl apply`
          # ...
          name: nginx
        ports:
          - containerPort: 80
            # ...
            # ...
            # ...
  # ...
```

Warning: Mixing `kubectl apply` with the imperative object configuration commands `create` and `replace` is not supported. This is because `create` and `replace` do not retain the `kubectl.kubernetes.io/last-applied-configuration` that `kubectl apply` uses to compute updates.

How to delete objects

There are two approaches to delete objects managed by `kubectl apply`.

Recommended: `kubectl delete -f <filename>`

Manually deleting objects using the imperative command is the recommended approach, as it is more explicit about what is being deleted, and less likely to result in the user deleting something unintentionally:

```
kubectl delete -f <filename>
```

Alternative: `kubectl apply -f <directory> --prune`

As an alternative to `kubectl delete`, you can use `kubectl apply` to identify objects to be deleted after their manifests have been removed from a directory in the local filesystem.

In Kubernetes 1.28, there are two pruning modes available in `kubectl apply`:

- Allowlist-based pruning: This mode has existed since `kubectl v1.5` but is still in alpha due to usability, correctness and performance issues with its design. The `ApplySet`-based mode is designed to replace it.
- `ApplySet`-based pruning: An *apply set* is a server-side object (by default, a `Secret`) that `kubectl` can use to accurately and efficiently track set membership across **apply** operations. This mode was introduced in alpha in `kubectl v1.27` as a replacement for allowlist-based pruning.

[Allow list](#)

[Apply set](#)

FEATURE STATE: `Kubernetes v1.5 [alpha]`

Warning: Take care when using `--prune` with `kubectl apply` in allow list mode. Which objects are pruned depends on the values of the `--prune-allowlist`, `--selector` and `--namespace` flags, and relies on dynamic discovery of the objects in scope. Especially if flag values are changed between invocations, this can lead to objects being unexpectedly deleted or retained.

To use allowlist-based pruning, add the following flags to your `kubectl apply` invocation:

- `--prune` : Delete previously applied objects that are not in the set passed to the current invocation.
- `--prune-allowlist` : A list of group-version-kinds (GVKs) to consider for pruning. This flag is optional but strongly encouraged, as its default value is a partial list of both namespaced and cluster-scoped types, which can lead to surprising results.
- `--selector/-l` : Use a label selector to constrain the set of objects selected for pruning. This flag is optional but strongly encouraged.
- `--all` : use instead of `--selector/-l` to explicitly select all previously applied objects of the allowlisted types.

Allowlist-based pruning queries the API server for all objects of the allowlisted GVKs that match the given labels (if any), and attempts to match the returned live object configurations against the object manifest files. If an object matches the query, and it does not have a manifest in the directory, and it has a `kubectl.kubernetes.io/last-applied-configuration` annotation, it is deleted.

```
kubectl apply -f <directory> --prune -l <labels> --prune-allowlist=<gvk-list>
```

Warning: Apply with prune should only be run against the root directory containing the object manifests. Running against sub-directories can cause objects to be unintentionally deleted if they were previously applied, have the labels given (if any), and do not appear in the subdirectory.

How to view an object

You can use `kubectl get` with `-o yaml` to view the configuration of a live object:

```
kubectl get -f <filename|url> -o yaml
```

How apply calculates differences and merges changes

Caution: A *patch* is an update operation that is scoped to specific fields of an object instead of the entire object. This enables updating only a specific set of fields on an object without reading the object first.

When `kubectl apply` updates the live configuration for an object, it does so by sending a patch request to the API server. The patch defines updates scoped to specific fields of the live object configuration. The `kubectl apply` command calculates this patch request using the configuration file, the live configuration, and the `last-applied-configuration` annotation stored in the live configuration.

Merge patch calculation

The `kubectl apply` command writes the contents of the configuration file to the `kubectl.kubernetes.io/last-applied-configuration` annotation. This is used to identify fields that have been removed from the configuration file and need to be cleared from the live configuration. Here are the steps used to calculate which fields should be deleted or set:

1. Calculate the fields to delete. These are the fields present in `last-applied-configuration` and missing from the configuration file.
2. Calculate the fields to add or set. These are the fields present in the configuration file whose values don't match the live configuration.

Here's an example. Suppose this is the configuration file for a Deployment object:

[application/update_deployment.yaml](#) 

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-deployment
spec:
  selector:
    matchLabels:
      app: nginx
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
        - name: nginx
          image: nginx:1.16.1 # update the image
          ports:
            - containerPort: 80
```

Also, suppose this is the live configuration for the same Deployment object:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  annotations:
    # ...
    # note that the annotation does not contain replicas
    # because it was not updated through apply
    kubectl.kubernetes.io/last-applied-configuration: |
      {"apiVersion": "apps/v1", "kind": "Deployment",
       "metadata": {"annotations": {}, "name": "nginx-deployment", "namespace": "default"}, "spec": {"minReadySeconds": 5, "selector": {"matchLabels": {"app": "nginx"}}, "template": {"spec": {"containers": [{"image": "nginx:1.14.2", "name": "nginx", "ports": [{"containerPort": 80}]}]}}, "status": {}}
    # ...
spec:
  replicas: 2 # written by scale
  # ...
  minReadySeconds: 5
  selector:
    matchLabels:
      # ...
      app: nginx
  template:
    metadata:
      # ...
      labels:
        app: nginx
    spec:
      containers:
        - image: nginx:1.14.2
          # ...
          name: nginx
        ports:
          - containerPort: 80
  # ...
```

Here are the merge calculations that would be performed by `kubectl apply`:

1. Calculate the fields to delete by reading values from `last-applied-configuration` and comparing them to values in the configuration file. Clear fields explicitly set to null in the local object configuration file regardless of whether they appear in the `last-applied-configuration`. In this example, `minReadySeconds` appears in the `last-applied-configuration` annotation, but does not appear in the configuration file. **Action:** Clear `minReadySeconds` from the live configuration.
2. Calculate the fields to set by reading values from the configuration file and comparing them to values in the live configuration. In this example, the value of `image` in the configuration file does not match the value in the live configuration. **Action:** Set the value of `image` in the live configuration.
3. Set the `last-applied-configuration` annotation to match the value of the configuration file.
4. Merge the results from 1, 2, 3 into a single patch request to the API server.

Here is the live configuration that is the result of the merge:

```

apiVersion: apps/v1
kind: Deployment
metadata:
  annotations:
    # ...
    # The annotation contains the updated image to nginx 1.16.1,
    # but does not contain the updated replicas to 2
  kubectl.kubernetes.io/last-applied-configuration: |
    {"apiVersion": "apps/v1", "kind": "Deployment",
     "metadata": {"annotations": {}, "name": "nginx-deployment", "namespace": "default"}, "spec": {"selector": {"matchLabels": {"app": "nginx"}}, "template": {"metadata": {"labels": {"app": "nginx"}}, "spec": {"containers": [{"image": "nginx:1.16.1", "name": "nginx", "ports": [{"containerPort": 80}]}]}}}}
    # ...
spec:
  selector:
    matchLabels:
      # ...
      app: nginx
  replicas: 2 # Set by `kubectl scale`. Ignored by `kubectl apply`.
  # minReadySeconds cleared by `kubectl apply`
  # ...
  template:
    metadata:
      # ...
      labels:
        app: nginx
    spec:
      containers:
        - image: nginx:1.16.1 # Set by `kubectl apply`
          # ...
          name: nginx
        ports:
          - containerPort: 80
            # ...
            # ...
        # ...
      # ...
# ...

```

How different types of fields are merged

How a particular field in a configuration file is merged with the live configuration depends on the type of the field. There are several types of fields:

- *primitive*: A field of type string, integer, or boolean. For example, `image` and `replicas` are primitive fields. **Action**: Replace.
- *map*, also called *object*: A field of type map or a complex type that contains subfields. For example, `labels` , `annotations` , `spec` and `metadata` are all maps. **Action**: Merge elements or subfields.
- *list*: A field containing a list of items that can be either primitive types or maps. For example, `containers` , `ports` , and `args` are lists. **Action**: Varies.

When `kubectl apply` updates a map or list field, it typically does not replace the entire field, but instead updates the individual subelements. For instance, when merging the `spec` on a Deployment, the entire `spec` is not replaced. Instead the subfields of `spec` , such as `replicas` , are compared and merged.

Merging changes to primitive fields

Primitive fields are replaced or cleared.

Note: - is used for "not applicable" because the value is not used.

Field in object configuration file	Field in live object configuration	Field in last-applied-configuration	Action
Yes	Yes	-	Set live to configuration file value.
Yes	No	-	Set live to local configuration.
No	-	Yes	Clear from live configuration.
No	-	No	Do nothing. Keep live value.

Merging changes to map fields

Fields that represent maps are merged by comparing each of the subfields or elements of the map:

Note: - is used for "not applicable" because the value is not used.

Key in object configuration file	Key in live object configuration	Field in last-applied-configuration	Action
Yes	Yes	-	Compare sub fields values.
Yes	No	-	Set live to local configuration.
No	-	Yes	Delete from live configuration.
No	-	No	Do nothing. Keep live value.

Merging changes for fields of type list

Merging changes to a list uses one of three strategies:

- Replace the list if all its elements are primitives.
- Merge individual elements in a list of complex elements.
- Merge a list of primitive elements.

The choice of strategy is made on a per-field basis.

Replace the list if all its elements are primitives

Treat the list the same as a primitive field. Replace or delete the entire list.
This preserves ordering.

Example: Use `kubectl apply` to update the `args` field of a Container in a Pod. This sets the value of `args` in the live configuration to the value in the configuration file. Any `args` elements that had previously been added to the live configuration are lost. The order of the `args` elements defined in the configuration file is retained in the live configuration.

```
# Last-applied-configuration value
args: ["a", "b"]

# configuration file value
args: ["a", "c"]

# Live configuration
args: ["a", "b", "d"]

# result after merge
args: ["a", "c"]
```

Explanation: The merge used the configuration file value as the new list value.

Merge individual elements of a list of complex elements:

Treat the list as a map, and treat a specific field of each element as a key. Add, delete, or update individual elements. This does not preserve ordering.

This merge strategy uses a special tag on each field called a `patchMergeKey`. The `patchMergeKey` is defined for each field in the Kubernetes source code: [types.go](#) When merging a list of maps, the field specified as the `patchMergeKey` for a given element is used like a map key for that element.

Example: Use `kubectl apply` to update the `containers` field of a PodSpec. This merges the list as though it was a map where each element is keyed by `name`.

```
# Last-applied-configuration value
containers:
- name: nginx
  image: nginx:1.16
- name: nginx-helper-a # key: nginx-helper-a; will be deleted in result
  image: helper:1.3
- name: nginx-helper-b # key: nginx-helper-b; will be retained
  image: helper:1.3

# configuration file value
containers:
- name: nginx
  image: nginx:1.16
- name: nginx-helper-b
  image: helper:1.3
- name: nginx-helper-c # key: nginx-helper-c; will be added in result
  image: helper:1.3

# Live configuration
containers:
- name: nginx
  image: nginx:1.16
- name: nginx-helper-a
  image: helper:1.3
- name: nginx-helper-b
  image: helper:1.3
  args: ["run"] # Field will be retained
- name: nginx-helper-d # key: nginx-helper-d; will be retained
  image: helper:1.3

# result after merge
containers:
- name: nginx
  image: nginx:1.16
  # Element nginx-helper-a was deleted
- name: nginx-helper-b
  image: helper:1.3
  args: ["run"] # Field was retained
- name: nginx-helper-c # Element was added
  image: helper:1.3
- name: nginx-helper-d # Element was ignored
  image: helper:1.3
```

Explanation:

- The container named "nginx-helper-a" was deleted because no container named "nginx-helper-a" appeared in the configuration file.
- The container named "nginx-helper-b" retained the changes to `args` in the live configuration. `kubectl apply` was able to identify that "nginx-helper-b" in the live configuration was the same "nginx-helper-b" as in the configuration file, even though their fields had different values (no `args` in the configuration file). This is because the `patchMergeKey` field value (`name`) was identical in both.
- The container named "nginx-helper-c" was added because no container with that name appeared in the live configuration, but one with that name appeared in the configuration file.
- The container named "nginx-helper-d" was retained because no element with that name appeared in the last-applied-configuration.

Merge a list of primitive elements

As of Kubernetes 1.5, merging lists of primitive elements is not supported.

Note: Which of the above strategies is chosen for a given field is controlled by the `patchStrategy` tag in [types.go](#). If no `patchStrategy` is specified for a field of type list, then the list is replaced.

Default field values

The API server sets certain fields to default values in the live configuration if they are not specified when the object is created.

Here's a configuration file for a Deployment. The file does not specify strategy :

[application/simple_deployment.yaml](#) 

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-deployment
spec:
  selector:
    matchLabels:
      app: nginx
  minReadySeconds: 5
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
        - name: nginx
          image: nginx:1.14.2
          ports:
            - containerPort: 80
```

Create the object using `kubectl apply` :

```
kubectl apply -f https://k8s.io/examples/application/simple_deployment
```

Print the live configuration using `kubectl get` :

```
kubectl get -f https://k8s.io/examples/application/simple_deployment.ya
```

The output shows that the API server set several fields to default values in the live configuration. These fields were not specified in the configuration file.

```
apiVersion: apps/v1
kind: Deployment
# ...
spec:
  selector:
    matchLabels:
      app: nginx
  minReadySeconds: 5
  replicas: 1 # defaulted by apiserver
  strategy:
    rollingUpdate: # defaulted by apiserver - derived from strategy.type
    maxSurge: 1
    maxUnavailable: 1
  type: RollingUpdate # defaulted by apiserver
  template:
    metadata:
      creationTimestamp: null
      labels:
        app: nginx
    spec:
      containers:
        - image: nginx:1.14.2
          imagePullPolicy: IfNotPresent # defaulted by apiserver
          name: nginx
          ports:
            - containerPort: 80
              protocol: TCP # defaulted by apiserver
          resources: {} # defaulted by apiserver
          terminationMessagePath: /dev/termination-log # defaulted by ap
      dnsPolicy: ClusterFirst # defaulted by apiserver
      restartPolicy: Always # defaulted by apiserver
      securityContext: {} # defaulted by apiserver
      terminationGracePeriodSeconds: 30 # defaulted by apiserver
# ...
```

In a patch request, defaulted fields are not re-defaulted unless they are explicitly cleared as part of a patch request. This can cause unexpected behavior for fields that are defaulted based on the values of other fields. When the other fields are later changed, the values defaulted from them will not be updated unless they are explicitly cleared.

For this reason, it is recommended that certain fields defaulted by the server are explicitly defined in the configuration file, even if the desired values match the server defaults. This makes it easier to recognize conflicting values that will not be re-defaulted by the server.

Example:

```
# Last-applied-configuration
spec:
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
        - name: nginx
          image: nginx:1.14.2
          ports:
            - containerPort: 80

# configuration file
spec:
  strategy:
    type: Recreate # updated value
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
        - name: nginx
          image: nginx:1.14.2
          ports:
            - containerPort: 80

# Live configuration
spec:
  strategy:
    type: RollingUpdate # defaulted value
    rollingUpdate: # defaulted value derived from type
      maxSurge : 1
      maxUnavailable: 1
  template:
    metadata:
      labels:
        app: nginx
  spec:
    containers:
      - name: nginx
        image: nginx:1.14.2
        ports:
          - containerPort: 80

# result after merge - ERROR!
spec:
  strategy:
    type: Recreate # updated value: incompatible with rollingUpdate
    rollingUpdate: # defaulted value: incompatible with "type: Recreate"
      maxSurge : 1
      maxUnavailable: 1
  template:
    metadata:
      labels:
        app: nginx
  spec:
    containers:
      - name: nginx
        image: nginx:1.14.2
        ports:
          - containerPort: 80
```

Explanation:

1. The user creates a Deployment without defining `strategy.type`.
2. The server defaults `strategy.type` to `RollingUpdate` and defaults the `strategy.rollingUpdate` values.
3. The user changes `strategy.type` to `Recreate`. The `strategy.rollingUpdate` values remain at their defaulted values, though the server expects them to be cleared. If the `strategy.rollingUpdate` values had been defined initially in the configuration file, it would have been more clear that they needed to be deleted.
4. Apply fails because `strategy.rollingUpdate` is not cleared. The `strategy.rollingUpdate` field cannot be defined with a `strategy.type` of `Recreate`.

Recommendation: These fields should be explicitly defined in the object configuration file:

- Selectors and PodTemplate labels on workloads, such as Deployment, StatefulSet, Job, DaemonSet, ReplicaSet, and ReplicationController
- Deployment rollout strategy

How to clear server-defaulted fields or fields set by other writers

Fields that do not appear in the configuration file can be cleared by setting their values to `null` and then applying the configuration file. For fields defaulted by the server, this triggers re-defaulting the values.

How to change ownership of a field between the configuration file and direct imperative writers

These are the only methods you should use to change an individual object field:

- Use `kubectl apply`.
- Write directly to the live configuration without modifying the configuration file: for example, use `kubectl scale`.

Changing the owner from a direct imperative writer to a configuration file

Add the field to the configuration file. For the field, discontinue direct updates to the live configuration that do not go through `kubectl apply`.

Changing the owner from a configuration file to a direct imperative writer

As of Kubernetes 1.5, changing ownership of a field from a configuration file to an imperative writer requires manual steps:

- Remove the field from the configuration file.
- Remove the field from the `kubectl.kubernetes.io/last-applied-configuration` annotation on the live object.

Changing management methods

Kubernetes objects should be managed using only one method at a time.

Switching from one method to another is possible, but is a manual process.

Note: It is OK to use imperative deletion with declarative management.

Migrating from imperative command management to declarative object configuration

Migrating from imperative command management to declarative object configuration involves several manual steps:

1. Export the live object to a local configuration file:

```
kubectl get <kind>/<name> -o yaml > <kind>_<name>.yaml
```

2. Manually remove the `status` field from the configuration file.

Note: This step is optional, as `kubectl apply` does not update the `status` field even if it is present in the configuration file.

3. Set the `kubectl.kubernetes.io/last-applied-configuration` annotation on the object:

```
kubectl replace --save-config -f <kind>_<name>.yaml
```

4. Change processes to use `kubectl apply` for managing the object exclusively.

Migrating from imperative object configuration to declarative object configuration

1. Set the `kubectl.kubernetes.io/last-applied-configuration` annotation on the object:

```
kubectl replace --save-config -f <kind>_<name>.yaml
```

2. Change processes to use `kubectl apply` for managing the object exclusively.

Defining controller selectors and PodTemplate labels

Warning: Updating selectors on controllers is strongly discouraged.

The recommended approach is to define a single, immutable PodTemplate label used only by the controller selector with no other semantic meaning.

Example:

```
selector:  
  matchLabels:  
    controller-selector: "apps/v1/deployment/nginx"  
template:  
  metadata:  
    labels:  
      controller-selector: "apps/v1/deployment/nginx"
```

What's next

- [Managing Kubernetes Objects Using Imperative Commands](#)
- [Imperative Management of Kubernetes Objects Using Configuration Files](#)
- [Kubectl Command Reference](#)
- [Kubernetes API Reference](#)

5.2 - Declarative Management of Kubernetes Objects Using Kustomize

Kustomize is a standalone tool to customize Kubernetes objects through a [kustomization file](#).

Since 1.14, Kubectl also supports the management of Kubernetes objects using a kustomization file. To view Resources found in a directory containing a kustomization file, run the following command:

```
kubectl kustomize <kustomization_directory>
```

To apply those Resources, run `kubectl apply` with `--kustomize` or `-k` flag:

```
kubectl apply -k <kustomization_directory>
```

Before you begin

Install [kubectl](#).

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

- [Killercoda](#)
- [Play with Kubernetes](#)

To check the version, enter `kubectl version`.

Overview of Kustomize

Kustomize is a tool for customizing Kubernetes configurations. It has the following features to manage application configuration files:

- generating resources from other sources
- setting cross-cutting fields for resources
- composing and customizing collections of resources

Generating Resources

ConfigMaps and Secrets hold configuration or sensitive data that are used by other Kubernetes objects, such as Pods. The source of truth of ConfigMaps or Secrets are usually external to a cluster, such as a `.properties` file or an SSH keyfile. Kustomize has `secretGenerator` and `configMapGenerator`, which generate Secret and ConfigMap from files or literals.

configMapGenerator

To generate a ConfigMap from a file, add an entry to the `files` list in `configMapGenerator`. Here is an example of generating a ConfigMap with a data item from a `.properties` file:

```
# Create a application.properties file
cat <<EOF >application.properties
FOO=Bar
EOF

cat <<EOF >./kustomization.yaml
configMapGenerator:
- name: example-configmap-1
  files:
  - application.properties
EOF
```

The generated ConfigMap can be examined with the following command:

```
kubectl kustomize ./
```

The generated ConfigMap is:

```
apiVersion: v1
data:
  application.properties: |
    FOO=Bar
kind: ConfigMap
metadata:
  name: example-configmap-1-8mbdf7882g
```

To generate a ConfigMap from an env file, add an entry to the `envs` list in `configMapGenerator`. Here is an example of generating a ConfigMap with a data item from a `.env` file:

```
# Create a .env file
cat <<EOF >.env
FOO=Bar
EOF

cat <<EOF >./kustomization.yaml
configMapGenerator:
- name: example-configmap-1
  envs:
  - .env
EOF
```

The generated ConfigMap can be examined with the following command:

```
kubectl kustomize ./
```

The generated ConfigMap is:

```
apiVersion: v1
data:
  FOO: Bar
kind: ConfigMap
metadata:
  name: example-configmap-1-42cfbf598f
```

Note: Each variable in the `.env` file becomes a separate key in the ConfigMap that you generate. This is different from the previous example which embeds a file named `application.properties` (and all its entries) as the value for a single key.

ConfigMaps can also be generated from literal key-value pairs. To generate a ConfigMap from a literal key-value pair, add an entry to the `literals` list in `configMapGenerator`. Here is an example of generating a ConfigMap with a data item from a key-value pair:

```
cat <<EOF ./kustomization.yaml
configMapGenerator:
- name: example-configmap-2
  literals:
  - FOO=Bar
EOF
```

The generated ConfigMap can be checked by the following command:

```
kubectl kustomize ./
```

The generated ConfigMap is:

```
apiVersion: v1
data:
  FOO: Bar
kind: ConfigMap
metadata:
  name: example-configmap-2-g2hdhfc6tk
```

To use a generated ConfigMap in a Deployment, reference it by the name of the `configMapGenerator`. Kustomize will automatically replace this name with the generated name.

This is an example deployment that uses a generated ConfigMap:

```
# Create a application.properties file
cat <<EOF >application.properties
FOO=Bar
EOF

cat <<EOF >deployment.yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: my-app
  labels:
    app: my-app
spec:
  selector:
    matchLabels:
      app: my-app
  template:
    metadata:
      labels:
        app: my-app
    spec:
      containers:
        - name: app
          image: my-app
          volumeMounts:
            - name: config
              mountPath: /config
      volumes:
        - name: config
          configMap:
            name: example-configmap-1
EOF
```

```
cat <<EOF >./kustomization.yaml
resources:
- deployment.yaml
configMapGenerator:
- name: example-configmap-1
  files:
    - application.properties
EOF
```

Generate the ConfigMap and Deployment:

```
kubectl kustomize ./
```

The generated Deployment will refer to the generated ConfigMap by name:

```
apiVersion: v1
data:
  application.properties: |
    FOO=Bar
kind: ConfigMap
metadata:
  name: example-configmap-1-g4hk9g2ff8
---
apiVersion: apps/v1
kind: Deployment
metadata:
  labels:
    app: my-app
  name: my-app
spec:
  selector:
    matchLabels:
      app: my-app
  template:
    metadata:
      labels:
        app: my-app
    spec:
      containers:
        - image: my-app
          name: app
        volumeMounts:
          - mountPath: /config
            name: config
      volumes:
        - configMap:
            name: example-configmap-1-g4hk9g2ff8
            name: config
```

secretGenerator

You can generate Secrets from files or literal key-value pairs. To generate a Secret from a file, add an entry to the `files` list in `secretGenerator`. Here is an example of generating a Secret with a data item from a file:

```
# Create a password.txt file
cat <<EOF >./password.txt
username=admin
password=secret
EOF

cat <<EOF >./kustomization.yaml
secretGenerator:
  - name: example-secret-1
    files:
      - password.txt
EOF
```

The generated Secret is as follows:

```
apiVersion: v1
data:
  password.txt: dXNlcm5hbWU9YWRtaW4KcGFzc3dvcmQ9c2VjcmV0Cg==
kind: Secret
metadata:
  name: example-secret-1-t2kt65hgtb
type: Opaque
```

To generate a Secret from a literal key-value pair, add an entry to `literals` list in `secretGenerator`. Here is an example of generating a Secret with a data item from a key-value pair:

```
cat <<EOF >./kustomization.yaml
secretGenerator:
- name: example-secret-2
  literals:
  - username=admin
  - password=secret
EOF
```

The generated Secret is as follows:

```
apiVersion: v1
data:
  password: c2VjcmV0
  username: YWRtaW4=
kind: Secret
metadata:
  name: example-secret-2-t52t6g96d8
type: Opaque
```

Like ConfigMaps, generated Secrets can be used in Deployments by referring to the name of the `secretGenerator`:

```
# Create a password.txt file
cat <<EOF >./password.txt
username=admin
password=secret
EOF

cat <<EOF >deployment.yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: my-app
  labels:
    app: my-app
spec:
  selector:
    matchLabels:
      app: my-app
  template:
    metadata:
      labels:
        app: my-app
    spec:
      containers:
        - name: app
          image: my-app
          volumeMounts:
            - name: password
              mountPath: /secrets
      volumes:
        - name: password
          secret:
            secretName: example-secret-1
EOF

cat <<EOF >./kustomization.yaml
resources:
- deployment.yaml
secretGenerator:
- name: example-secret-1
  files:
    - password.txt
EOF
```

generatorOptions

The generated ConfigMaps and Secrets have a content hash suffix appended. This ensures that a new ConfigMap or Secret is generated when the contents are changed. To disable the behavior of appending a suffix, one can use `generatorOptions`. Besides that, it is also possible to specify cross-cutting options for generated ConfigMaps and Secrets.

```
cat <<EOF >./kustomization.yaml
configMapGenerator:
- name: example-configmap-3
  literals:
    - FOO=Bar
generatorOptions:
  disableNameSuffixHash: true
  labels:
    type: generated
  annotations:
    note: generated
EOF
```

Run `kubectl kustomize ./` to view the generated ConfigMap:

```
apiVersion: v1
data:
  FOO: Bar
kind: ConfigMap
metadata:
  annotations:
    note: generated
  labels:
    type: generated
  name: example-configmap-3
```

Setting cross-cutting fields

It is quite common to set cross-cutting fields for all Kubernetes resources in a project. Some use cases for setting cross-cutting fields:

- setting the same namespace for all Resources
- adding the same name prefix or suffix
- adding the same set of labels
- adding the same set of annotations

Here is an example:

```
# Create a deployment.yaml
cat <<EOF >./deployment.yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-deployment
  labels:
    app: nginx
spec:
  selector:
    matchLabels:
      app: nginx
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
        - name: nginx
          image: nginx
EOF

cat <<EOF >./kustomization.yaml
namespace: my-namespace
namePrefix: dev-
nameSuffix: "-001"
commonLabels:
  app: bingo
commonAnnotations:
  oncallPager: 800-555-1212
resources:
  - deployment.yaml
EOF
```

Run `kubectl kustomize ./` to view those fields are all set in the Deployment Resource:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  annotations:
    oncallPager: 800-555-1212
  labels:
    app: bingo
  name: dev-nginx-deployment-001
  namespace: my-namespace
spec:
  selector:
    matchLabels:
      app: bingo
  template:
    metadata:
      annotations:
        oncallPager: 800-555-1212
      labels:
        app: bingo
    spec:
      containers:
        - image: nginx
          name: nginx
```

Composing and Customizing Resources

It is common to compose a set of Resources in a project and manage them inside the same file or directory. Kustomize offers composing Resources from different files and applying patches or other customization to them.

Composing

Kustomize supports composition of different resources. The `resources` field, in the `kustomization.yaml` file, defines the list of resources to include in a configuration. Set the path to a resource's configuration file in the `resources` list. Here is an example of an NGINX application comprised of a Deployment and a Service:

```
# Create a deployment.yaml file
cat <<EOF > deployment.yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: my-nginx
spec:
  selector:
    matchLabels:
      run: my-nginx
  replicas: 2
  template:
    metadata:
      labels:
        run: my-nginx
    spec:
      containers:
        - name: my-nginx
          image: nginx
          ports:
            - containerPort: 80
EOF

# Create a service.yaml file
cat <<EOF > service.yaml
apiVersion: v1
kind: Service
metadata:
  name: my-nginx
  labels:
    run: my-nginx
spec:
  ports:
    - port: 80
      protocol: TCP
  selector:
    run: my-nginx
EOF

# Create a kustomization.yaml composing them
cat <<EOF >/kustomization.yaml
resources:
  - deployment.yaml
  - service.yaml
EOF
```

The Resources from `kubectl kustomize ./` contain both the Deployment and the Service objects.

Customizing

Patches can be used to apply different customizations to Resources.

Kustomize supports different patching mechanisms through

`patchesStrategicMerge` and `patchesJson6902`. `patchesStrategicMerge` is a list of file paths. Each file should be resolved to a [strategic merge patch](#). The names inside the patches must match Resource names that are already loaded. Small patches that do one thing are recommended. For example, create one patch for increasing the deployment replica number and another patch for setting the memory limit.

```
# Create a deployment.yaml file
cat <<EOF > deployment.yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: my-nginx
spec:
  selector:
    matchLabels:
      run: my-nginx
  replicas: 2
  template:
    metadata:
      labels:
        run: my-nginx
    spec:
      containers:
        - name: my-nginx
          image: nginx
          ports:
            - containerPort: 80
EOF

# Create a patch increase_replicas.yaml
cat <<EOF > increase_replicas.yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: my-nginx
spec:
  replicas: 3
EOF

# Create another patch set_memory.yaml
cat <<EOF > set_memory.yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: my-nginx
spec:
  template:
    spec:
      containers:
        - name: my-nginx
          resources:
            limits:
              memory: 512Mi
EOF

cat <<EOF >./kustomization.yaml
resources:
- deployment.yaml
patchesStrategicMerge:
- increase_replicas.yaml
- set_memory.yaml
EOF
```

Run `kubectl kustomize ./` to view the Deployment:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: my-nginx
spec:
  replicas: 3
  selector:
    matchLabels:
      run: my-nginx
  template:
    metadata:
      labels:
        run: my-nginx
    spec:
      containers:
        - image: nginx
          name: my-nginx
          ports:
            - containerPort: 80
          resources:
            limits:
              memory: 512Mi
```

Not all Resources or fields support strategic merge patches. To support modifying arbitrary fields in arbitrary Resources, Kustomize offers applying [JSON patch](#) through `patchesJson6902`. To find the correct Resource for a Json patch, the group, version, kind and name of that Resource need to be specified in `kustomization.yaml`. For example, increasing the replica number of a Deployment object can also be done through `patchesJson6902`.

```
# Create a deployment.yaml file
cat <<EOF > deployment.yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: my-nginx
spec:
  selector:
    matchLabels:
      run: my-nginx
  replicas: 2
  template:
    metadata:
      labels:
        run: my-nginx
    spec:
      containers:
        - name: my-nginx
          image: nginx
          ports:
            - containerPort: 80
EOF
```

```
# Create a json patch
cat <<EOF > patch.yaml
- op: replace
  path: /spec/relicas
  value: 3
EOF
```

```
# Create a kustomization.yaml
cat <<EOF >./kustomization.yaml
resources:
- deployment.yaml

patchesJson6902:
- target:
    group: apps
    version: v1
    kind: Deployment
    name: my-nginx
    path: patch.yaml
EOF
```

Run `kubectl kustomize ./` to see the `replicas` field is updated:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: my-nginx
spec:
  replicas: 3
  selector:
    matchLabels:
      run: my-nginx
  template:
    metadata:
      labels:
        run: my-nginx
    spec:
      containers:
        - image: nginx
          name: my-nginx
          ports:
            - containerPort: 80
```

In addition to patches, Kustomize also offers customizing container images or injecting field values from other objects into containers without creating patches. For example, you can change the image used inside containers by specifying the new image in `images` field in `kustomization.yaml`.

```
cat <<EOF > deployment.yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: my-nginx
spec:
  selector:
    matchLabels:
      run: my-nginx
  replicas: 2
  template:
    metadata:
      labels:
        run: my-nginx
    spec:
      containers:
        - name: my-nginx
          image: nginx
          ports:
            - containerPort: 80
EOF

cat <<EOF >./kustomization.yaml
resources:
- deployment.yaml
images:
- name: nginx
  newName: my.image.registry/nginx
  newTag: 1.4.0
EOF
```

Run `kubectl kustomize ./` to see that the image being used is updated:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: my-nginx
spec:
  replicas: 2
  selector:
    matchLabels:
      run: my-nginx
  template:
    metadata:
      labels:
        run: my-nginx
    spec:
      containers:
        - image: my.image.registry/nginx:1.4.0
          name: my-nginx
          ports:
            - containerPort: 80
```

Sometimes, the application running in a Pod may need to use configuration values from other objects. For example, a Pod from a Deployment object need to read the corresponding Service name from Env or as a command argument. Since the Service name may change as `namePrefix` or `nameSuffix` is added in the `kustomization.yaml` file. It is not

recommended to hard code the Service name in the command argument.
For this usage, Kustomize can inject the Service name into containers
through `vars`.

```
# Create a deployment.yaml file (quoting the here doc delimiter)
cat <<'EOF' > deployment.yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: my-nginx
spec:
  selector:
    matchLabels:
      run: my-nginx
  replicas: 2
  template:
    metadata:
      labels:
        run: my-nginx
    spec:
      containers:
        - name: my-nginx
          image: nginx
          command: ["start", "--host", "$(MY_SERVICE_NAME)"]
EOF

# Create a service.yaml file
cat <<EOF > service.yaml
apiVersion: v1
kind: Service
metadata:
  name: my-nginx
  labels:
    run: my-nginx
spec:
  ports:
    - port: 80
      protocol: TCP
  selector:
    run: my-nginx
EOF

cat <<EOF >./kustomization.yaml
namePrefix: dev-
nameSuffix: "-001"

resources:
  - deployment.yaml
  - service.yaml

vars:
  - name: MY_SERVICE_NAME
    objref:
      kind: Service
      name: my-nginx
      apiVersion: v1
EOF
```

Run `kubectl kustomize ./` to see that the Service name injected into
containers is `dev-my-nginx-001`:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: dev-my-nginx-001
spec:
  replicas: 2
  selector:
    matchLabels:
      run: my-nginx
  template:
    metadata:
      labels:
        run: my-nginx
    spec:
      containers:
        - command:
          - start
          - --host
          - dev-my-nginx-001
        image: nginx
        name: my-nginx
```

Bases and Overlays

Kustomize has the concepts of **bases** and **overlays**. A **base** is a directory with a `kustomization.yaml`, which contains a set of resources and associated customization. A base could be either a local directory or a directory from a remote repo, as long as a `kustomization.yaml` is present inside. An **overlay** is a directory with a `kustomization.yaml` that refers to other kustomization directories as its `bases`. A **base** has no knowledge of an overlay and can be used in multiple overlays. An overlay may have multiple bases and it composes all resources from bases and may also have customization on top of them.

Here is an example of a base:

```
# Create a directory to hold the base
mkdir base

# Create a base/deployment.yaml
cat <<EOF > base/deployment.yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: my-nginx
spec:
  selector:
    matchLabels:
      run: my-nginx
  replicas: 2
  template:
    metadata:
      labels:
        run: my-nginx
    spec:
      containers:
        - name: my-nginx
          image: nginx
EOF

# Create a base/service.yaml file
cat <<EOF > base/service.yaml
apiVersion: v1
kind: Service
metadata:
  name: my-nginx
  labels:
    run: my-nginx
spec:
  ports:
    - port: 80
      protocol: TCP
  selector:
    run: my-nginx
EOF

# Create a base/kustomization.yaml
cat <<EOF > base/kustomization.yaml
resources:
  - deployment.yaml
  - service.yaml
EOF
```

This base can be used in multiple overlays. You can add different `namePrefix` or other cross-cutting fields in different overlays. Here are two overlays using the same base.

```
mkdir dev
cat <<EOF > dev/kustomization.yaml
resources:
  - ../base
namePrefix: dev-
EOF

mkdir prod
cat <<EOF > prod/kustomization.yaml
resources:
  - ../base
namePrefix: prod-
EOF
```

How to apply/view/delete objects using Kustomize

Use `--kustomize` or `-k` in `kubectl` commands to recognize Resources managed by `kustomization.yaml`. Note that `-k` should point to a kustomization directory, such as

```
kubectl apply -k <kustomization directory>/
```

Given the following `kustomization.yaml`,

```
# Create a deployment.yaml file
cat <<EOF > deployment.yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: my-nginx
spec:
  selector:
    matchLabels:
      run: my-nginx
  replicas: 2
  template:
    metadata:
      labels:
        run: my-nginx
    spec:
      containers:
        - name: my-nginx
          image: nginx
          ports:
            - containerPort: 80
EOF

# Create a kustomization.yaml
cat <<EOF >./kustomization.yaml
namePrefix: dev-
commonLabels:
  app: my-nginx
resources:
- deployment.yaml
EOF
```

Run the following command to apply the Deployment object `dev-my-nginx`:

```
> kubectl apply -k ./
deployment.apps/dev-my-nginx created
```

Run one of the following commands to view the Deployment object `dev-my-nginx`:

```
kubectl get -k ./
```

```
kubectl describe -k ./
```

Run the following command to compare the Deployment object `dev-my-nginx` against the state that the cluster would be in if the manifest was applied:

```
kubectl diff -k ./
```

Run the following command to delete the Deployment object `dev-my-nginx`:

```
> kubectl delete -k ./  
deployment.apps "dev-my-nginx" deleted
```

Kustomize Feature List

Field	Type	Explanation
namespace	string	add namespace to all resources
namePrefix	string	value of this field is prepended to the names of all resources
nameSuffix	string	value of this field is appended to the names of all resources
commonLabels	map[string]string	labels to add to all resources and selectors
commonAnnotations	map[string]string	annotations to add to all resources
resources	[]string	each entry in this list must resolve to an existing resource configuration file
configMapGenerator	[] ConfigMapArgs	Each entry in this list generates a ConfigMap
secretGenerator	[] SecretArgs	Each entry in this list generates a Secret
generatorOptions	GeneratorOptions	Modify behaviors of all ConfigMap and Secret generator
bases	[]string	Each entry in this list should resolve to a directory containing a <code>kustomization.yaml</code> file
patchesStrategicMerge	[]string	Each entry in this list should resolve a strategic merge patch of a Kubernetes object
patchesJson6902	[] Patch	Each entry in this list should resolve to a Kubernetes object and a Json Patch
vars	[] Var	Each entry is to capture text from one resource's field
images	[] Image	Each entry is to modify the name, tags and/or digest for one image without creating patches

Field	Type	Explanation
configurations	[]string	Each entry in this list should resolve to a file containing Kustomize transformer configurations
crds	[]string	Each entry in this list should resolve to an OpenAPI definition file for Kubernetes types

What's next

- [Kustomize](#)
- [Kubectl Book](#)
- [Kubectl Command Reference](#)
- [Kubernetes API Reference](#)

5.3 - Managing Kubernetes Objects Using Imperative Commands

Kubernetes objects can quickly be created, updated, and deleted directly using imperative commands built into the `kubectl` command-line tool. This document explains how those commands are organized and how to use them to manage live objects.

Before you begin

Install [kubectl](#).

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

- [KillerCoda](#)
- [Play with Kubernetes](#)

To check the version, enter `kubectl version`.

Trade-offs

The `kubectl` tool supports three kinds of object management:

- Imperative commands
- Imperative object configuration
- Declarative object configuration

See [Kubernetes Object Management](#) for a discussion of the advantages and disadvantage of each kind of object management.

How to create objects

The `kubectl` tool supports verb-driven commands for creating some of the most common object types. The commands are named to be recognizable to users unfamiliar with the Kubernetes object types.

- `run` : Create a new Pod to run a Container.
- `expose` : Create a new Service object to load balance traffic across Pods.
- `autoscale` : Create a new Autoscaler object to automatically horizontally scale a controller, such as a Deployment.

The `kubectl` tool also supports creation commands driven by object type. These commands support more object types and are more explicit about their intent, but require users to know the type of objects they intend to create.

- `create <objecttype> [<subtype>] <instancename>`

Some objects types have subtypes that you can specify in the `create` command. For example, the Service object has several subtypes including ClusterIP, LoadBalancer, and NodePort. Here's an example that creates a Service with subtype NodePort:

```
kubectl create service nodeport <myservicename>
```

In the preceding example, the `create service nodeport` command is called a subcommand of the `create service` command.

You can use the `-h` flag to find the arguments and flags supported by a subcommand:

```
kubectl create service nodeport -h
```

How to update objects

The `kubectl` command supports verb-driven commands for some common update operations. These commands are named to enable users unfamiliar with Kubernetes objects to perform updates without knowing the specific fields that must be set:

- `scale` : Horizontally scale a controller to add or remove Pods by updating the replica count of the controller.
- `annotate` : Add or remove an annotation from an object.
- `label` : Add or remove a label from an object.

The `kubectl` command also supports update commands driven by an aspect of the object. Setting this aspect may set different fields for different object types:

- `set <field>` : Set an aspect of an object.

Note: In Kubernetes version 1.5, not every verb-driven command has an associated aspect-driven command.

The `kubectl` tool supports these additional ways to update a live object directly, however they require a better understanding of the Kubernetes object schema.

- `edit` : Directly edit the raw configuration of a live object by opening its configuration in an editor.
- `patch` : Directly modify specific fields of a live object by using a patch string. For more details on patch strings, see the patch section in [API Conventions](#).

How to delete objects

You can use the `delete` command to delete an object from a cluster:

- `delete <type>/<name>`

Note: You can use `kubectl delete` for both imperative commands and imperative object configuration. The difference is in the arguments passed to the command. To use `kubectl delete` as an imperative command, pass the object to be deleted as an argument. Here's an example that passes a Deployment object named nginx:

```
kubectl delete deployment/nginx
```

How to view an object

There are several commands for printing information about an object:

- `get` : Prints basic information about matching objects. Use `get -h` to see a list of options.
- `describe` : Prints aggregated detailed information about matching objects.
- `logs` : Prints the stdout and stderr for a container running in a Pod.

Using `set` commands to modify objects before creation

There are some object fields that don't have a flag you can use in a `create` command. In some of those cases, you can use a combination of `set` and `create` to specify a value for the field before object creation. This is done by piping the output of the `create` command to the `set` command, and then back to the `create` command. Here's an example:

```
kubectl create service clusterip my-svc --clusterip="None" -o yaml --d
```

1. The `kubectl create service -o yaml --dry-run=client` command creates the configuration for the Service, but prints it to stdout as YAML instead of sending it to the Kubernetes API server.
2. The `kubectl set selector --local -f - -o yaml` command reads the configuration from stdin, and writes the updated configuration to stdout as YAML.
3. The `kubectl create -f -` command creates the object using the configuration provided via stdin.

Using `--edit` to modify objects before creation

You can use `kubectl create --edit` to make arbitrary changes to an object before it is created. Here's an example:

```
kubectl create service clusterip my-svc --clusterip="None" -o yaml --d
kubectl create --edit -f /tmp/srv.yaml
```

1. The `kubectl create service` command creates the configuration for the Service and saves it to `/tmp/srv.yaml`.
2. The `kubectl create --edit` command opens the configuration file for editing before it creates the object.

What's next

- [Imperative Management of Kubernetes Objects Using Configuration Files](#)

- [Declarative Management of Kubernetes Objects Using Configuration Files](#)
- [Kubectl Command Reference](#)
- [Kubernetes API Reference](#)

5.4 - Imperative Management of Kubernetes Objects Using Configuration Files

Kubernetes objects can be created, updated, and deleted by using the `kubectl` command-line tool along with an object configuration file written in YAML or JSON. This document explains how to define and manage objects using configuration files.

Before you begin

Install [kubectl](#).

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

- [KillerCoda](#)
- [Play with Kubernetes](#)

To check the version, enter `kubectl version`.

Trade-offs

The `kubectl` tool supports three kinds of object management:

- Imperative commands
- Imperative object configuration
- Declarative object configuration

See [Kubernetes Object Management](#) for a discussion of the advantages and disadvantage of each kind of object management.

How to create objects

You can use `kubectl create -f` to create an object from a configuration file. Refer to the [Kubernetes API reference](#) for details.

- `kubectl create -f <filename|url>`

How to update objects

Warning: Updating objects with the `replace` command drops all parts of the spec not specified in the configuration file. This should not be used with objects whose specs are partially managed by the cluster, such as Services of type `LoadBalancer`, where the `externalIPs` field is managed independently from the configuration file. Independently managed fields must be copied to the configuration file to prevent `replace` from dropping them.

You can use `kubectl replace -f` to update a live object according to a configuration file.

- `kubectl replace -f <filename|url>`

How to delete objects

You can use `kubectl delete -f` to delete an object that is described in a configuration file.

- `kubectl delete -f <filename|url>`

Note:

If configuration file has specified the `generateName` field in the `metadata` section instead of the `name` field, you cannot delete the object using `kubectl delete -f <filename|url>`. You will have to use other flags for deleting the object. For example:

```
kubectl delete <type> <name>
kubectl delete <type> -l <label>
```

How to view an object

You can use `kubectl get -f` to view information about an object that is described in a configuration file.

- `kubectl get -f <filename|url> -o yaml`

The `-o yaml` flag specifies that the full object configuration is printed. Use `kubectl get -h` to see a list of options.

Limitations

The `create`, `replace`, and `delete` commands work well when each object's configuration is fully defined and recorded in its configuration file. However when a live object is updated, and the updates are not merged into its configuration file, the updates will be lost the next time a `replace` is executed. This can happen if a controller, such as a HorizontalPodAutoscaler, makes updates directly to a live object. Here's an example:

1. You create an object from a configuration file.
2. Another source updates the object by changing some field.
3. You replace the object from the configuration file. Changes made by the other source in step 2 are lost.

If you need to support multiple writers to the same object, you can use `kubectl apply` to manage the object.

Creating and editing an object from a URL without saving the configuration

Suppose you have the URL of an object configuration file. You can use `kubectl create --edit` to make changes to the configuration before the object is created. This is particularly useful for tutorials and tasks that point to a configuration file that could be modified by the reader.

```
kubectl create -f <url> --edit
```

Migrating from imperative commands to imperative object configuration

Migrating from imperative commands to imperative object configuration involves several manual steps.

1. Export the live object to a local object configuration file:

```
kubectl get <kind>/<name> -o yaml > <kind>_<name>.yaml
```

2. Manually remove the status field from the object configuration file.
3. For subsequent object management, use `replace` exclusively.

```
kubectl replace -f <kind>_<name>.yaml
```

Defining controller selectors and PodTemplate labels

Warning: Updating selectors on controllers is strongly discouraged.

The recommended approach is to define a single, immutable PodTemplate label used only by the controller selector with no other semantic meaning.

Example label:

```
selector:  
  matchLabels:  
    controller-selector: "apps/v1/deployment/nginx"  
template:  
  metadata:  
    labels:  
      controller-selector: "apps/v1/deployment/nginx"
```

What's next

- [Managing Kubernetes Objects Using Imperative Commands](#)
- [Declarative Management of Kubernetes Objects Using Configuration Files](#)
- [Kubectl Command Reference](#)
- [Kubernetes API Reference](#)

5.5 - Update API Objects in Place Using kubectl patch

Use kubectl patch to update Kubernetes API objects in place.

Do a strategic merge patch or a JSON merge 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

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

- [KillerCoda](#)
- [Play with Kubernetes](#)

To check the version, enter `kubectl version`.

Use a strategic merge patch to update a Deployment

Here's the configuration file for a Deployment that has two replicas. Each replica is a Pod that has one container:

[application/deployment-patch.yaml](#) 

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: patch-demo
spec:
  replicas: 2
  selector:
    matchLabels:
      app: nginx
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
        - name: patch-demo-ctr
          image: nginx
      tolerations:
        - effect: NoSchedule
          key: dedicated
          value: test-team
```

Create the Deployment:

```
kubectl apply -f https://k8s.io/examples/application/deployment-patch.
```

View the Pods associated with your Deployment:

```
kubectl get pods
```

The output shows that the Deployment has two Pods. The 1/1 indicates that each Pod has one container:

NAME	READY	STATUS	RESTARTS	AGE
patch-demo-28633765-670qr	1/1	Running	0	23s
patch-demo-28633765-j5qs3	1/1	Running	0	23s

Make a note of the names of the running Pods. Later, you will see that these Pods get terminated and replaced by new ones.

At this point, each Pod has one Container that runs the nginx image. Now suppose you want each Pod to have two containers: one that runs nginx and one that runs redis.

Create a file named `patch-file.yaml` that has this content:

```
spec:  
  template:  
    spec:  
      containers:  
        - name: patch-demo-ctr-2  
          image: redis
```

Patch your Deployment:

```
kubectl patch deployment patch-demo --patch-file patch-file.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" patchType:"strategic"`
    ...
}
```

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-file patch-file-tolerance
```

View the patched Deployment:

```
kubectl get deployment patch-demo --output yaml
```

The output shows that the PodSpec in the Deployment has only one Tolerance:

```
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,1,rep,name=Tolerations"`
    ...
}
```

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-file patch-fi
```

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

Use strategic merge patch to update a Deployment using the retainKeys strategy

Here's the configuration file for a Deployment that uses the `RollingUpdate` strategy:

[application/deployment-retainkeys.yaml](#) 

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: retainkeys-demo
spec:
  selector:
    matchLabels:
      app: nginx
  strategy:
    rollingUpdate:
      maxSurge: 30%
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
        - name: retainkeys-demo-ctr
          image: nginx
```

Create the deployment:

```
kubectl apply -f https://k8s.io/examples/application/deployment-retain
```

At this point, the deployment is created and is using the `RollingUpdate` strategy.

Create a file named `patch-file-no-retainkeys.yaml` that has this content:

```
spec:  
  strategy:  
    type: Recreate
```

Patch your Deployment:

```
kubectl patch deployment retainkeys-demo --type strategic --patch-file
```

In the output, you can see that it is not possible to set `type` as `Recreate` when a value is defined for `spec.strategy.rollingUpdate`:

```
The Deployment "retainkeys-demo" is invalid: spec.strategy.rollingUpdat
```

The way to remove the value for `spec.strategy.rollingUpdate` when updating the value for `type` is to use the `retainKeys` strategy for the strategic merge.

Create another file named `patch-file-retainkeys.yaml` that has this content:

```
spec:  
  strategy:  
    $retainKeys:  
      - type  
    type: Recreate
```

With this patch, we indicate that we want to retain only the `type` key of the `strategy` object. Thus, the `rollingUpdate` will be removed during the patch operation.

Patch your Deployment again with this new patch:

```
kubectl patch deployment retainkeys-demo --type strategic --patch-file
```

Examine the content of the Deployment:

```
kubectl get deployment retainkeys-demo --output yaml
```

The output shows that the `strategy` object in the Deployment does not contain the `rollingUpdate` key anymore:

```
spec:  
  strategy:  
    type: Recreate  
  template:
```

Notes on the strategic merge patch using the retainKeys strategy

The patch you did in the preceding exercise is called a *strategic merge patch with retainKeys strategy*. This method introduces a new directive `$retainKeys` that has the following strategies:

- It contains a list of strings.
- All fields needing to be preserved must be present in the `$retainKeys` list.
- The fields that are present will be merged with live object.
- All of the missing fields will be cleared when patching.
- All fields in the `$retainKeys` list must be a superset or the same as the fields present in the patch.

The `retainKeys` strategy does not work for all objects. It only works when the value of the `patchStrategy` key in a field tag in the Kubernetes source code contains `retainKeys`. For example, the `Strategy` field of the `DeploymentSpec` struct has a `patchStrategy` of `retainKeys`:

```
type DeploymentSpec struct {
    ...
    // +patchStrategy=retainKeys
    Strategy DeploymentStrategy `json:"strategy,omitempty" patchStrategy
    ...
}
```

You can also see the `retainKeys` strategy in the [OpenAPI spec](#):

```
"io.k8s.api.apps.v1.DeploymentSpec": {
    ...
    "strategy": {
        "$ref": "#/definitions/io.k8s.api.apps.v1.DeploymentStrategy",
        "description": "The deployment strategy to use to replace existing pods when deploying a new version of the application",
        "x-kubernetes-patch-strategy": "retainKeys"
    },
    ...
}
```

And you can see the `retainKeys` strategy in the [Kubernetes API documentation](#).

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-file patch-file.yaml
kubectl patch deployment patch-demo --patch 'spec:\n  template:\n    spec'

kubectl patch deployment patch-demo --patch-file patch-file.json
kubectl patch deployment patch-demo --patch '{"spec": {"template": {"s'}
```

Update an object's replica count using `kubectl patch` with `--subresource`

FEATURE STATE: `Kubernetes v1.24 [alpha]`

The flag `--subresource=[subresource-name]` is used with `kubectl` commands like `get`, `patch`, `edit` and `replace` to fetch and update `status` and `scale` subresources of the resources (applicable for `kubectl` version v1.24 or more). This flag is used with all the API resources (built-in and CRs) that have `status` or `scale` subresource. Deployment is one of the examples which supports these subresources.

Here's a manifest for a Deployment that has two replicas:

[application/deployment.yaml](#) 

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-deployment
spec:
  selector:
    matchLabels:
      app: nginx
  replicas: 2 # tells deployment to run 2 pods matching the template
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
        - name: nginx
          image: nginx:1.14.2
        ports:
          - containerPort: 80
```

Create the Deployment:

```
kubectl apply -f https://k8s.io/examples/application/deployment.yaml
```

View the Pods associated with your Deployment:

```
kubectl get pods -l app=nginx
```

In the output, you can see that Deployment has two Pods. For example:

NAME	READY	STATUS	RESTARTS	AGE
nginx-deployment-7fb96c846b-22567	1/1	Running	0	47s
nginx-deployment-7fb96c846b-mlgns	1/1	Running	0	47s

Now, patch that Deployment with `--subresource=[subresource-name]` flag:

```
kubectl patch deployment nginx-deployment --subresource='scale' --type
```

The output is:

```
scale.autoscaling/nginx-deployment patched
```

View the Pods associated with your patched Deployment:

```
kubectl get pods -l app=nginx
```

In the output, you can see one new pod is created, so now you have 3 running pods.

NAME	READY	STATUS	RESTARTS	AGE
nginx-deployment-7fb96c846b-22567	1/1	Running	0	107s
nginx-deployment-7fb96c846b-lxfr2	1/1	Running	0	14s
nginx-deployment-7fb96c846b-mlgns	1/1	Running	0	107s

View the patched Deployment:

```
kubectl get deployment nginx-deployment -o yaml
```

```
...
spec:
  replicas: 3
...
status:
...
  availableReplicas: 3
  readyReplicas: 3
  replicas: 3
```

Note: If you run `kubectl patch` and specify `--subresource` flag for resource that doesn't support that particular subresource, the API server returns a 404 Not Found error.

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](#).

Note: Strategic merge patch is not supported for custom resources.

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](#)

6 - Managing Secrets

Managing confidential settings data using Secrets.

6.1 - Managing Secrets using kubectl

Creating Secret objects using kubectl command line.

This page shows you how to create, edit, manage, and delete Kubernetes Secrets using the `kubectl` command-line tool.

Before you begin

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

- [KillerCoda](#)
- [Play with Kubernetes](#)

Create a Secret

A `Secret` object stores sensitive data such as credentials used by Pods to access services. For example, you might need a Secret to store the username and password needed to access a database.

You can create the Secret by passing the raw data in the command, or by storing the credentials in files that you pass in the command. The following commands create a Secret that stores the username `admin` and the password `S!B*d$zDsb=`.

Use raw data

Run the following command:

```
kubectl create secret generic db-user-pass \
--from-literal=username=admin \
--from-literal=password='S!B\*d$zDsb='
```

You must use single quotes `''` to escape special characters such as `$`, `\`, `*`, `=`, and `!` in your strings. If you don't, your shell will interpret these characters.

Use source files

1. Store the credentials in files:

```
echo -n 'admin' > ./username.txt
echo -n 'S!B\*d$zDsb=' > ./password.txt
```

The `-n` flag ensures that the generated files do not have an extra newline character at the end of the text. This is important because when `kubectl` reads a file and encodes the content into a base64 string, the extra newline character gets encoded too. You do not need to escape special characters in strings that you include in a file.

2. Pass the file paths in the `kubectl` command:

```
kubectl create secret generic db-user-pass \
--from-file=./username.txt \
--from-file=./password.txt
```

The default key name is the file name. You can optionally set the key name using `--from-file=[key=]source`. For example:

```
kubectl create secret generic db-user-pass \
--from-file=username=./username.txt \
--from-file=password=./password.txt
```

With either method, the output is similar to:

```
secret/db-user-pass created
```

Verify the Secret

Check that the Secret was created:

```
kubectl get secrets
```

The output is similar to:

NAME	TYPE	DATA	AGE
db-user-pass	Opaque	2	51s

View the details of the Secret:

```
kubectl describe secret db-user-pass
```

The output is similar to:

```
Name: db-user-pass
Namespace: default
Labels: <none>
Annotations: <none>

Type: Opaque

Data
=====
password: 12 bytes
username: 5 bytes
```

The commands `kubectl get` and `kubectl describe` avoid showing the contents of a `Secret` by default. This is to protect the `Secret` from being exposed accidentally, or from being stored in a terminal log.

Decode the Secret

1. View the contents of the Secret you created:

```
kubectl get secret db-user-pass -o jsonpath='{.data}'
```

The output is similar to:

```
{ "password": "UyFCXCpkJHpEc2I9", "username": "YWRtaW4=" }
```

2. Decode the password data:

```
echo 'UyFCXCpkJHpEc2I9' | base64 --decode
```

The output is similar to:

```
S!B\*d$zDsb=
```

Caution: This is an example for documentation purposes. In practice, this method could cause the command with the encoded data to be stored in your shell history. Anyone with access to your computer could find the command and decode the secret. A better approach is to combine the view and decode commands.

```
kubectl get secret db-user-pass -o jsonpath='{.data.password}' |
```

Edit a Secret

You can edit an existing `Secret` object unless it is [immutable](#). To edit a Secret, run the following command:

```
kubectl edit secrets <secret-name>
```

This opens your default editor and allows you to update the base64 encoded Secret values in the `data` field, such as in the following example:

```
# Please edit the object below. Lines beginning with a '#' will be ignored
# and an empty file will abort the edit. If an error occurs while saving,
# reopened with the relevant failures.
#
apiVersion: v1
data:
  password: UyFCXCpkJHpEc2I9
  username: YWRtaW4=
kind: Secret
metadata:
  creationTimestamp: "2022-06-28T17:44:13Z"
  name: db-user-pass
  namespace: default
  resourceVersion: "12708504"
  uid: 91becd59-78fa-4c85-823f-6d44436242ac
type: Opaque
```

Clean up

To delete a Secret, run the following command:

```
kubectl delete secret db-user-pass
```

What's next

- Read more about the [Secret concept](#)
- Learn how to [manage Secrets using config file](#)
- Learn how to [manage Secrets using kustomize](#)

6.2 - Managing Secrets using Configuration File

Creating Secret objects using resource configuration file.

Before you begin

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

- [KillerCoda](#)
- [Play with Kubernetes](#)

Create the Secret

You can define the `Secret` object in a manifest first, in JSON or YAML format, and then create that object. The [Secret](#) resource contains two maps: `data` and `stringData`. The `data` field is used to store arbitrary data, encoded using base64. The `stringData` field is provided for convenience, and it allows you to provide the same data as unencoded strings. The keys of `data` and `stringData` must consist of alphanumeric characters, `-`, `_` or `.`

The following example stores two strings in a Secret using the `data` field.

1. Convert the strings to base64:

```
echo -n 'admin' | base64  
echo -n '1f2d1e2e67df' | base64
```

Note: The serialized JSON and YAML values of Secret data are encoded as base64 strings. Newlines are not valid within these strings and must be omitted. When using the `base64` utility on Darwin/macOS, users should avoid using the `-b` option to split long lines. Conversely, Linux users *should* add the option `-w 0` to `base64` commands or the pipeline `base64 | tr -d '\n'` if the `-w` option is not available.

The output is similar to:

```
YWRTaW4=  
MwYyZDF1MmU2N2Rm
```

2. Create the manifest:

```
apiVersion: v1
kind: Secret
metadata:
  name: mysecret
type: Opaque
data:
  username: YWRtaW4=
  password: MWYyZDF1MmU2N2Rm
```

Note that the name of a Secret object must be a valid [DNS subdomain name](#).

3. Create the Secret using `kubectl apply`:

```
kubectl apply -f ./secret.yaml
```

The output is similar to:

```
secret/mysecret created
```

To verify that the Secret was created and to decode the Secret data, refer to [Managing Secrets using kubectl](#).

Specify unencoded data when creating a Secret

For certain scenarios, you may wish to use the `stringData` field instead. This field allows you to put a non-base64 encoded string directly into the Secret, and the string will be encoded for you when the Secret is created or updated.

A practical example of this might be where you are deploying an application that uses a Secret to store a configuration file, and you want to populate parts of that configuration file during your deployment process.

For example, if your application uses the following configuration file:

```
apiUrl: "https://my.api.com/api/v1"
username: "<user>"
password: "<password>"
```

You could store this in a Secret using the following definition:

```
apiVersion: v1
kind: Secret
metadata:
  name: mysecret
type: Opaque
stringData:
  config.yaml: |
    apiUrl: "https://my.api.com/api/v1"
    username: <user>
    password: <password>
```

Note: The `stringData` field for a Secret does not work well with server-side apply.

When you retrieve the Secret data, the command returns the encoded values, and not the plaintext values you provided in `stringData`.

For example, if you run the following command:

```
kubectl get secret mysecret -o yaml
```

The output is similar to:

```
apiVersion: v1
data:
  config.yaml: YXBpVXJsOiAiaHR0cHM6Ly9teS5hcGkuY29tL2FwaS92MSIKdXNlcm5l
kind: Secret
metadata:
  creationTimestamp: 2018-11-15T20:40:59Z
  name: mysecret
  namespace: default
  resourceVersion: "7225"
  uid: c280ad2e-e916-11e8-98f2-025000000001
type: Opaque
```

Specify both `data` and `stringData`

If you specify a field in both `data` and `stringData`, the value from `stringData` is used.

For example, if you define the following Secret:

```
apiVersion: v1
kind: Secret
metadata:
  name: mysecret
type: Opaque
data:
  username: YWRtaW4=
stringData:
  username: administrator
```

Note: The `stringData` field for a Secret does not work well with server-side apply.

The `Secret` object is created as follows:

```
apiVersion: v1
data:
  username: YWRtaW5pc3RyYXRvcg==
kind: Secret
metadata:
  creationTimestamp: 2018-11-15T20:46:46Z
  name: mysecret
  namespace: default
  resourceVersion: "7579"
  uid: 91460ecb-e917-11e8-98f2-025000000001
type: Opaque
```

```
YWRTaw5pc3RyYXRvcg== decodes to administrator .
```

Edit a Secret

To edit the data in the Secret you created using a manifest, modify the `data` or `stringData` field in your manifest and apply the file to your cluster. You can edit an existing `Secret` object unless it is [immutable](#).

For example, if you want to change the password from the previous example to `birdsarentreal`, do the following:

1. Encode the new password string:

```
echo -n 'birdsarentreal' | base64
```

The output is similar to:

```
YmlyZHNhcmVudHJ1YWw=
```

2. Update the `data` field with your new password string:

```
apiVersion: v1
kind: Secret
metadata:
  name: mysecret
  type: Opaque
data:
  username: YWRtaW4=
  password: YmlyZHNhcmVudHJ1YWw=
```

3. Apply the manifest to your cluster:

```
kubectl apply -f ./secret.yaml
```

The output is similar to:

```
secret/mysecret configured
```

Kubernetes updates the existing `Secret` object. In detail, the `kubectl` tool notices that there is an existing `Secret` object with the same name.

`kubectl` fetches the existing object, plans changes to it, and submits the changed `Secret` object to your cluster control plane.

If you specified `kubectl apply --server-side` instead, `kubectl` uses [Server Side Apply](#) instead.

Clean up

To delete the Secret you have created:

```
kubectl delete secret mysecret
```

What's next

- Read more about the [Secret concept](#)
- Learn how to [manage Secrets using kubectl](#)
- Learn how to [manage Secrets using kustomize](#)

6.3 - Managing Secrets using Kustomize

Creating Secret objects using kustomization.yaml file.

kubectl supports using the [Kustomize object management tool](#) to manage Secrets and ConfigMaps. You create a *resource generator* using Kustomize, which generates a Secret that you can apply to the API server using kubectl .

Before you begin

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

- [Killercoda](#)
- [Play with Kubernetes](#)

Create a Secret

You can generate a Secret by defining a `secretGenerator` in a `kustomization.yaml` file that references other existing files, `.env` files, or literal values. For example, the following instructions create a Kustomization file for the username `admin` and the password `1f2d1e2e67df` .

Create the Kustomization file

Literals Files .env files

```
secretGenerator:  
- name: database-creds  
  literals:  
    - username=admin  
    - password=1f2d1e2e67df
```

In all cases, you don't need to base64 encode the values. The name of the YAML file **must** be `kustomization.yaml` OR `kustomization.yml` .

Apply the kustomization file

To create the Secret, apply the directory that contains the kustomization file:

```
kubectl apply -k <directory-path>
```

The output is similar to:

```
secret/database-creds-5hdh7hhgfk created
```

When a Secret is generated, the Secret name is created by hashing the Secret data and appending the hash value to the name. This ensures that a new Secret is generated each time the data is modified.

To verify that the Secret was created and to decode the Secret data,

```
kubectl get -k <directory-path> -o jsonpath='{.data}'
```

The output is similar to:

```
{ "password": "UyFCXCpkJHpEc2I9", "username": "YWRTaw4=" }
```

```
echo 'UyFCXCpkJHpEc2I9' | base64 --decode
```

The output is similar to:

```
S!B\*d$zDsb=
```

For more information, refer to [Managing Secrets using kubectl](#) and [Declarative Management of Kubernetes Objects Using Kustomize](#).

Edit a Secret

1. In your `kustomization.yaml` file, modify the data, such as the `password` .
2. Apply the directory that contains the kustomization file:

```
kubectl apply -k <directory-path>
```

The output is similar to:

```
secret/db-user-pass-6f24b56cc8 created
```

The edited Secret is created as a new `Secret` object, instead of updating the existing `Secret` object. You might need to update references to the Secret in your Pods.

Clean up

To delete a Secret, use `kubectl` :

```
kubectl delete secret db-user-pass
```

What's next

- Read more about the [Secret concept](#)
- Learn how to [manage Secrets using kubectl](#)
- Learn how to [manage Secrets using config file](#)

7 - Inject Data Into Applications

Specify configuration and other data for the Pods that run your workload.

7.1 - Define a Command and Arguments for a Container

This page shows how to define commands and arguments when you run a container in a Pod.

Before you begin

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

- [Killercoda](#)
- [Play with Kubernetes](#)

To check the version, enter `kubectl version`.

Define a command and arguments when you create a Pod

When you create a Pod, you can define a command and arguments for the containers that run in the Pod. To define a command, include the `command` field in the configuration file. To define arguments for the command, include the `args` field in the configuration file. The command and arguments that you define cannot be changed after the Pod is created.

The command and arguments that you define in the configuration file override the default command and arguments provided by the container image. If you define args, but do not define a command, the default command is used with your new arguments.

Note: The `command` field corresponds to `entrypoint` in some container runtimes.

In this exercise, you create a Pod that runs one container. The configuration file for the Pod defines a command and two arguments:

[pods/commands.yaml](#) 

```
apiVersion: v1
kind: Pod
metadata:
  name: command-demo
  labels:
    purpose: demonstrate-command
spec:
  containers:
    - name: command-demo-container
      image: debian
      command: ["printenv"]
      args: ["HOSTNAME", "KUBERNETES_PORT"]
  restartPolicy: OnFailure
```

1. Create a Pod based on the YAML configuration file:

```
kubectl apply -f https://k8s.io/examples/pods/commands.yaml
```

2. List the running Pods:

```
kubectl get pods
```

The output shows that the container that ran in the command-demo Pod has completed.

3. To see the output of the command that ran in the container, view the logs from the Pod:

```
kubectl logs command-demo
```

The output shows the values of the HOSTNAME and KUBERNETES_PORT environment variables:

```
command-demo
tcp://10.3.240.1:443
```

Use environment variables to define arguments

In the preceding example, you defined the arguments directly by providing strings. As an alternative to providing strings directly, you can define arguments by using environment variables:

```
env:
- name: MESSAGE
  value: "hello world"
  command: ["/bin/echo"]
  args: ["$(MESSAGE)"]
```

This means you can define an argument for a Pod using any of the techniques available for defining environment variables, including [ConfigMaps](#) and [Secrets](#).

Note: The environment variable appears in parentheses, "`$(VAR)`". This is required for the variable to be expanded in the `command` or `args` field.

Run a command in a shell

In some cases, you need your command to run in a shell. For example, your command might consist of several commands piped together, or it might be a shell script. To run your command in a shell, wrap it like this:

```
command: ["/bin/sh"]
args: ["-c", "while true; do echo hello; sleep 10;done"]
```

What's next

- Learn more about [configuring pods and containers](#).
- Learn more about [running commands in a container](#).
- See [Container](#).

7.2 - Define Dependent Environment Variables

This page shows how to define dependent environment variables for a container in a Kubernetes Pod.

Before you begin

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

- [Killercoda](#)
- [Play with Kubernetes](#)

Define an environment dependent variable for a container

When you create a Pod, you can set dependent environment variables for the containers that run in the Pod. To set dependent environment variables, you can use `$(VAR_NAME)` in the `value` of `env` in the configuration file.

In this exercise, you create a Pod that runs one container. The configuration file for the Pod defines a dependent environment variable with common usage defined. Here is the configuration manifest for the Pod:

[pods/inject/dependent-envvars.yaml](#) 

```
apiVersion: v1
kind: Pod
metadata:
  name: dependent-envvars-demo
spec:
  containers:
    - name: dependent-envvars-demo
      args:
        - while true; do echo -en '\n'; printf UNCHANGED_REFERENCE=$UN
      command:
        - sh
        - -c
      image: busybox:1.28
      env:
        - name: SERVICE_PORT
          value: "80"
        - name: SERVICE_IP
          value: "172.17.0.1"
        - name: UNCHANGED_REFERENCE
          value: "$(PROTOCOL)://$(SERVICE_IP):$(SERVICE_PORT)"
        - name: PROTOCOL
          value: "https"
        - name: SERVICE_ADDRESS
          value: "$(PROTOCOL)://$(SERVICE_IP):$(SERVICE_PORT)"
        - name: ESCAPED_REFERENCE
          value: "$$(PROTOCOL)://$(SERVICE_IP):$(SERVICE_PORT)"
```

1. Create a Pod based on that manifest:

```
kubectl apply -f https://k8s.io/examples/pods/inject/dependent-en
```

```
pod/dependent-envvars-demo created
```

2. List the running Pods:

```
kubectl get pods dependent-envvars-demo
```

NAME	READY	STATUS	RESTARTS	AGE
dependent-envvars-demo	1/1	Running	0	9s

3. Check the logs for the container running in your Pod:

```
kubectl logs pod/dependent-envvars-demo
```

```
UNCHANGED_REFERENCE=$(PROTOCOL)://172.17.0.1:80
SERVICE_ADDRESS=https://172.17.0.1:80
ESCAPED_REFERENCE=$((PROTOCOL)://172.17.0.1:80)
```

As shown above, you have defined the correct dependency reference of `SERVICE_ADDRESS`, bad dependency reference of `UNCHANGED_REFERENCE` and skip dependent references of `ESCAPED_REFERENCE`.

When an environment variable is already defined when being referenced, the reference can be correctly resolved, such as in the `SERVICE_ADDRESS` case.

Note that order matters in the `env` list. An environment variable is not considered "defined" if it is specified further down the list. That is why `UNCHANGED_REFERENCE` fails to resolve `$(PROTOCOL)` in the example above.

When the environment variable is undefined or only includes some variables, the undefined environment variable is treated as a normal string, such as `UNCHANGED_REFERENCE`. Note that incorrectly parsed environment variables, in general, will not block the container from starting.

The `$(VAR_NAME)` syntax can be escaped with a double `$`, ie: `$(VAR_NAME)`. Escaped references are never expanded, regardless of whether the referenced variable is defined or not. This can be seen from the `ESCAPED_REFERENCE` case above.

What's next

- Learn more about [environment variables](#).
- See [EnvVarSource](#).

7.3 - Define Environment Variables for a Container

This page shows how to define environment variables for a container in a Kubernetes Pod.

Before you begin

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

- [Killercoda](#)
- [Play with Kubernetes](#)

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.

The `env` and `envFrom` fields have different effects.

env

allows you to set environment variables for a container, specifying a value directly for each variable that you name.

envFrom

allows you to set environment variables for a container by referencing either a ConfigMap or a Secret. When you use `envFrom`, all the key-value pairs in the referenced ConfigMap or Secret are set as environment variables for the container. You can also specify a common prefix string.

You can read more about [ConfigMap](#) and [Secret](#).

This page explains how to use `env`.

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 manifest for the Pod:

[pods/inject/envvars.yaml](#) 

```
apiVersion: v1
kind: Pod
metadata:
  name: envar-demo
  labels:
    purpose: demonstrate-envvars
spec:
  containers:
    - name: envar-demo-container
      image: gcr.io/google-samples/node-hello:1.0
      env:
        - name: DEMO_GREETING
          value: "Hello from the environment"
        - name: DEMO_FAREWELL
          value: "Such a sweet sorrow"
```

1. Create a Pod based on that manifest:

```
kubectl apply -f https://k8s.io/examples/pods/inject/envvars.yaml
```

2. List the running Pods:

```
kubectl get pods -l purpose=demonstrate-envvars
```

The output is similar to:

NAME	READY	STATUS	RESTARTS	AGE
envar-demo	1/1	Running	0	9s

3. List the Pod's container environment variables:

```
kubectl exec 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
```

Note: The environment variables set using the `env` or `envFrom` field override any environment variables specified in the container image.

Note: Environment variables may reference each other, however ordering is important. Variables making use of others defined in the same context must come later in the list. Similarly, avoid circular references.

Using environment variables inside of your config

Environment variables that you define in a Pod's configuration under `.spec.containers[*].env[*]` can be used elsewhere in the configuration, for example in commands and arguments that you set for the Pod's containers. In the example configuration below, the `GREETING`, `HONORIFIC`, and `NAME` environment variables are set to `Warm greetings to`, `The Most Honorable`, and `Kubernetes`, respectively. The environment variable `MESSAGE` combines the set of all these environment variables and then uses it as a CLI argument passed to the `env-print-demo` container.

```
apiVersion: v1
kind: Pod
metadata:
  name: print-greeting
spec:
  containers:
    - name: env-print-demo
      image: bash
      env:
        - name: GREETING
          value: "Warm greetings to"
        - name: HONORIFIC
          value: "The Most Honorable"
        - name: NAME
          value: "Kubernetes"
        - name: MESSAGE
          value: "$(GREETING) $(HONORIFIC) $(NAME)"
      command: ["echo"]
      args: ["$(MESSAGE)"]
```

Upon creation, the command `echo Warm greetings to The Most Honorable Kubernetes` is run on the container.

What's next

- Learn more about [environment variables](#).
- Learn about [using secrets as environment variables](#).
- See [EnvVarSource](#).

7.4 - 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, using the *downward API*. You can use environment variables to expose Pod fields, container fields, or both.

In Kubernetes, there are two ways to expose Pod and container fields to a running container:

- *Environment variables*, as explained in this task
- [Volume files](#)

Together, these two ways of exposing Pod and container fields are called the downward API.

As Services are the primary mode of communication between containerized applications managed by Kubernetes, it is helpful to be able to discover them at runtime.

Read more about accessing Services [here](#).

Before you begin

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

- [KillerCoda](#)
- [Play with Kubernetes](#)

Use Pod fields as values for environment variables

In this part of exercise, you create a Pod that has one container, and you project Pod-level fields into the running container as environment variables.

[pods/inject/dapi-envvars-pod.yaml](#) 

```
apiVersion: v1
kind: Pod
metadata:
  name: dapi-envars-fieldref
spec:
  containers:
    - name: test-container
      image: registry.k8s.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 that manifest, you can see five environment variables. The `env` field is an array of environment variable definitions. The first element in the array specifies that the `MY_NODE_NAME` environment variable gets its value from the Pod's `spec.nodeName` field. Similarly, the other environment variables get their names from Pod fields.

Note: The fields in this example are Pod fields. They are not fields of the container in the Pod.

Create the Pod:

```
kubectl apply -f https://k8s.io/examples/pods/inject/dapi-envars-pod.yaml
```

Verify that the container in the Pod is running:

```
# If the new Pod isn't yet healthy, rerun this command a few times.
kubectl get pods
```

View the container's logs:

```
kubectl logs dapi-envars-fieldref
```

The output shows the values of selected environment variables:

```
minikube
dapi-envars-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-envars-fieldref -- sh
```

In your shell, view the environment variables:

```
# Run this in a shell inside the container
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-envars-fieldref
```

Use container fields as values for environment variables

In the preceding exercise, you used information from Pod-level fields as the values for environment variables. In this next exercise, you are going to pass fields that are part of the Pod definition, but taken from the specific [container](#) rather than from the Pod overall.

Here is a manifest for another Pod that again has just one container:

[pods/inject/dapi-envars-container.yaml](#) 

```
apiVersion: v1
kind: Pod
metadata:
  name: dapi-envars-resourcefieldref
spec:
  containers:
    - name: test-container
      image: registry.k8s.io/busybox:1.24
      command: [ "sh", "-c" ]
      args:
        - while true; do
          echo -en '\n';
          printenv MY_CPU_REQUEST MY_CPU_LIMIT;
          printenv MY_MEM_REQUEST MY_MEM_LIMIT;
          sleep 10;
        done;
  resources:
    requests:
      memory: "32Mi"
      cpu: "125m"
    limits:
      memory: "64Mi"
      cpu: "250m"
  env:
    - name: MY_CPU_REQUEST
      valueFrom:
        resourceFieldRef:
          containerName: test-container
          resource: requests.cpu
    - name: MY_CPU_LIMIT
      valueFrom:
        resourceFieldRef:
          containerName: test-container
          resource: limits.cpu
    - name: MY_MEM_REQUEST
      valueFrom:
        resourceFieldRef:
          containerName: test-container
          resource: requests.memory
    - name: MY_MEM_LIMIT
      valueFrom:
        resourceFieldRef:
          containerName: test-container
          resource: limits.memory
  restartPolicy: Never
```

In this manifest, you can see four environment variables. The `env` field is an array of environment variable definitions. 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 fields that are specific to this container.

Create the Pod:

```
kubectl apply -f https://k8s.io/examples/pods/inject/dapi-envars-conta
```

Verify that the container in the Pod is running:

```
# If the new Pod isn't yet healthy, rerun this command a few times.  
kubectl get pods
```

View the container's logs:

```
kubectl logs dapi-envvars-resourcefieldref
```

The output shows the values of selected environment variables:

```
1  
1  
33554432  
67108864
```

What's next

- Read [Defining Environment Variables for a Container](#)
- Read the [spec](#) API definition for Pod. This includes the definition of Container (part of Pod).
- Read the list of [available fields](#) that you can expose using the downward API.

Read about Pods, containers and environment variables in the legacy API reference:

- [PodSpec](#)
- [Container](#)
- [EnvVar](#)
- [EnvVarSource](#)
- [ObjectFieldSelector](#)
- [ResourceFieldSelector](#)

7.5 - Expose Pod Information to Containers Through Files

This page shows how a Pod can use a [downwardAPI volume](#), to expose information about itself to containers running in the Pod. A downwardAPI volume can expose Pod fields and container fields.

In Kubernetes, there are two ways to expose Pod and container fields to a running container:

- [Environment variables](#)
- Volume files, as explained in this task

Together, these two ways of exposing Pod and container fields are called the *downward API*.

Before you begin

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

- [KillerCoda](#)
- [Play with Kubernetes](#)

Store Pod fields

In this part of exercise, you create a Pod that has one container, and you project Pod-level fields into the running container as files. Here is the manifest for the Pod:

[pods/inject/dapi-volume.yaml](#) 

```
apiVersion: v1
kind: Pod
metadata:
  name: kubernetes-downwardapi-volume-example
  labels:
    zone: us-est-coast
    cluster: test-cluster1
    rack: rack-22
  annotations:
    build: two
    builder: john-doe
spec:
  containers:
    - name: client-container
      image: registry.k8s.io/busybox
      command: ["sh", "-c"]
      args:
        - while true; do
            if [[ -e /etc/podinfo/labels ]]; then
              echo -en '\n\n'; cat /etc/podinfo/labels; fi;
            if [[ -e /etc/podinfo/annotations ]]; then
              echo -en '\n\n'; cat /etc/podinfo/annotations; fi;
            sleep 5;
          done;
      volumeMounts:
        - name: podinfo
          mountPath: /etc/podinfo
  volumes:
    - name: podinfo
      downwardAPI:
        items:
          - path: "labels"
            fieldRef:
              fieldPath: metadata.labels
          - path: "annotations"
            fieldRef:
              fieldPath: metadata.annotations
```

In the manifest, 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 defines a `downwardAPI` volume. The first element specifies that the value of the Pod's `metadata.labels` field should be stored in a file named `labels`. The second element specifies that the value of the Pod's `annotations` field should be stored in a file named `annotations`.

Note: The fields in this example are Pod fields. They are not fields of the container in the Pod.

Create the Pod:

```
kubectl apply -f https://k8s.io/examples/pods/inject/dapi-volume.yaml
```

Verify that the container in the Pod is running:

```
kubectl get pods
```

View the container's logs:

```
kubectl logs kubernetes-downwardapi-volume-example
```

The output shows the contents of the `labels` file and the `annotations` file:

```
cluster="test-cluster1"
rack="rack-22"
zone="us-est-coast"

build="two"
builder="john-doe"
```

Get a shell into the container that is running in your Pod:

```
kubectl exec -it kubernetes-downwardapi-volume-example -- sh
```

In your shell, view the `labels` file:

```
/# cat /etc/podinfo/labels
```

The output shows that all of the Pod's labels have been written to the `labels` file:

```
cluster="test-cluster1"
rack="rack-22"
zone="us-est-coast"
```

Similarly, view the `annotations` file:

```
/# cat /etc/podinfo/annotations
```

View the files in the `/etc/podinfo` directory:

```
/# ls -laR /etc/podinfo
```

In the output, you can see that the `labels` and `annotations` files are in a temporary subdirectory: in this example, `..2982_06_02_21_47_53.299460680`. In the `/etc/podinfo` directory, `..data` is a symbolic link to the temporary subdirectory. Also in the `/etc/podinfo` directory, `labels` and `annotations` are symbolic links.

```
drwxr-xr-x  ... Feb 6 21:47 ..2982_06_02_21_47_53.299460680
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 made Pod-level fields accessible using the downward API. In this next exercise, you are going to pass fields that are part of the Pod definition, but taken from the specific [container](#) rather than from the Pod overall. Here is a manifest for a Pod that again has just one container:

[pods/inject/dapi-volume-resources.yaml](#) 

```
apiVersion: v1
kind: Pod
metadata:
  name: kubernetes-downwardapi-volume-example-2
spec:
  containers:
    - name: client-container
      image: registry.k8s.io/busybox:1.24
      command: ["sh", "-c"]
      args:
        - while true; do
          echo -en '\n';
          if [[ -e /etc/podinfo/cpu_limit ]]; then
            echo -en '\n'; cat /etc/podinfo/cpu_limit; fi;
          if [[ -e /etc/podinfo/cpu_request ]]; then
            echo -en '\n'; cat /etc/podinfo/cpu_request; fi;
          if [[ -e /etc/podinfo/mem_limit ]]; then
            echo -en '\n'; cat /etc/podinfo/mem_limit; fi;
          if [[ -e /etc/podinfo/mem_request ]]; then
            echo -en '\n'; cat /etc/podinfo/mem_request; fi;
          sleep 5;
        done;
  resources:
    requests:
      memory: "32Mi"
      cpu: "125m"
    limits:
      memory: "64Mi"
      cpu: "250m"
  volumeMounts:
    - name: podinfo
      mountPath: /etc/podinfo
  volumes:
    - name: podinfo
      downwardAPI:
        items:
          - path: "cpu_limit"
            resourceFieldRef:
              containerName: client-container
              resource: limits.cpu
              divisor: 1m
          - path: "cpu_request"
            resourceFieldRef:
              containerName: client-container
              resource: requests.cpu
              divisor: 1m
          - path: "mem_limit"
            resourceFieldRef:
              containerName: client-container
              resource: limits.memory
              divisor: 1Mi
          - path: "mem_request"
            resourceFieldRef:
              containerName: client-container
              resource: requests.memory
              divisor: 1Mi
```

In the manifest, you can see that the Pod has a [downwardAPI volume](#), and that the single container in that Pod mounts the volume at `/etc/podinfo`.

Look at the `items` array under `downwardAPI`. Each element of the array defines a file in the downward API volume.

The first element specifies that in the container named `client-container`, the value of the `limits.cpu` field in the format specified by `1m` should be published as a file named `cpu_limit`. The `divisor` field is optional and has the default value of `1`. A divisor of 1 means cores for `cpu` resources, or bytes for `memory` resources.

Create the Pod:

```
kubectl apply -f https://k8s.io/examples/pods/inject/dapi-volume-resou
```

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:

```
# Run this in a shell inside the container
cat /etc/podinfo/cpu_limit
```

You can use similar commands to view the `cpu_request`, `mem_limit` and `mem_request` files.

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](#).

What's next

- Read the [spec](#) API definition for Pod. This includes the definition of Container (part of Pod).
- Read the list of [available fields](#) that you can expose using the downward API.

Read about volumes in the legacy API reference:

- Check the [Volume](#) API definition which defines a generic volume in a Pod for containers to access.
- Check the [DownwardAPIVolumeSource](#) API definition which defines a volume that contains Downward API information.
- Check the [DownwardAPIVolumeFile](#) API definition which contains references to object or resource fields for populating a file in the Downward API volume.
- Check the [ResourceFieldSelector](#) API definition which specifies the container resources and their output format.

7.6 - 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

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

- [Killercoda](#)
- [Play with Kubernetes](#)

Convert your secret data to a base-64 representation

Suppose you want to have two pieces of secret data: a username `my-app` and a password `39528$vdg7Jb`. First, use a base64 encoding tool to convert your username and password to a base64 representation. Here's an example using the commonly available base64 program:

```
echo -n 'my-app' | base64  
echo -n '39528$vdg7Jb' | base64
```

The output shows that the base-64 representation of your username is `bXktYXBw`, and the base-64 representation of your password is `Mzk1MjgkdmRnN0pi`.

Caution: Use a local tool trusted by your OS to decrease the security risks of external tools.

Create a Secret

Here is a configuration file you can use to create a Secret that holds your username and password:

[pods/inject/secret.yaml](#) 

```
apiVersion: v1  
kind: Secret  
metadata:  
  name: test-secret  
data:  
  username: bXktYXBw  
  password: Mzk1MjgkdmRnN0pi
```

1. Create the Secret

```
kubectl apply -f https://k8s.io/examples/pods/inject/secret.yaml
```

2. View information about the Secret:

```
kubectl get secret test-secret
```

Output:

NAME	TYPE	DATA	AGE
test-secret	Opaque	2	1m

3. View more detailed information about the Secret:

```
kubectl describe secret test-secret
```

Output:

```
Name:      test-secret
Namespace: default
Labels:    <none>
Annotations: <none>

Type:     Opaque

Data
=====
password: 13 bytes
username: 7 bytes
```

Create a Secret directly with kubectl

If you want to skip the Base64 encoding step, you can create the same Secret using the `kubectl create secret` command. For example:

```
kubectl create secret generic test-secret --from-literal='username=my-
```

This is more convenient. The detailed approach shown earlier runs through each step explicitly to demonstrate what is happening.

Create a Pod that has access to the secret data through a Volume

Here is a configuration file you can use to create a Pod:

[pods/inject/secret-pod.yaml](#) 

```
apiVersion: v1
kind: Pod
metadata:
  name: secret-test-pod
spec:
  containers:
    - name: test-container
      image: nginx
      volumeMounts:
        # name must match the volume name below
        - name: secret-volume
          mountPath: /etc/secret-volume
          readOnly: true
      # The secret data is exposed to Containers in the Pod through a Volume
  volumes:
    - name: secret-volume
      secret:
        secretName: test-secret
```

1. Create the Pod:

```
kubectl apply -f https://k8s.io/examples/pods/inject/secret-pod.yaml
```

2. Verify that your Pod is running:

```
kubectl get pod secret-test-pod
```

Output:

NAME	READY	STATUS	RESTARTS	AGE
secret-test-pod	1/1	Running	0	42m

3. Get a shell into the Container that is running in your Pod:

```
kubectl exec -i -t 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, list the files in the `/etc/secret-volume` directory:

```
# Run this in the shell inside the container
ls /etc/secret-volume
```

The output shows two files, one for each piece of secret data:

```
password username
```

5. In your shell, display the contents of the `username` and `password` files:

```
# Run this in the shell inside the container
echo "$( cat /etc/secret-volume/username )"
echo "$( cat /etc/secret-volume/password )"
```

The output is your username and password:

```
my-app
39528$vdg7Jb
```

Modify your image or command line so that the program looks for files in the `mountPath` directory. Each key in the Secret `data` map becomes a file name in this directory.

Project Secret keys to specific file paths

You can also control the paths within the volume where Secret keys are projected. Use the `.spec.volumes[].secret.items` field to change the target path of each key:

```
apiVersion: v1
kind: Pod
metadata:
  name: mypod
spec:
  containers:
    - name: mypod
      image: redis
      volumeMounts:
        - name: foo
          mountPath: "/etc/foo"
          readOnly: true
  volumes:
    - name: foo
      secret:
        secretName: mysecret
        items:
          - key: username
            path: my-group/my-username
```

When you deploy this Pod, the following happens:

- The `username` key from `mysecret` is available to the container at the path `/etc/foo/my-group/my-username` instead of at `/etc/foo/username`.
- The `password` key from that Secret object is not projected.

If you list keys explicitly using `.spec.volumes[].secret.items`, consider the following:

- Only keys specified in `items` are projected.
- To consume all keys from the Secret, all of them must be listed in the `items` field.
- All listed keys must exist in the corresponding Secret. Otherwise, the volume is not created.

Set POSIX permissions for Secret keys

You can set the POSIX file access permission bits for a single Secret key. If you don't specify any permissions, `0644` is used by default. You can also set a default POSIX file mode for the entire Secret volume, and you can override per key if needed.

For example, you can specify a default mode like this:

```
apiVersion: v1
kind: Pod
metadata:
  name: mypod
spec:
  containers:
  - name: mypod
    image: redis
    volumeMounts:
    - name: foo
      mountPath: "/etc/foo"
  volumes:
  - name: foo
    secret:
      secretName: mysecret
    defaultMode: 0400
```

The Secret is mounted on `/etc/foo` ; all the files created by the secret volume mount have permission `0400` .

Note: If you're defining a Pod or a Pod template using JSON, beware that the JSON specification doesn't support octal literals for numbers because JSON considers `0400` to be the *decimal* value `400`. In JSON, use decimal values for the `defaultMode` instead. If you're writing YAML, you can write the `defaultMode` in octal.

Define container environment variables using Secret data

You can consume the data in Secrets as environment variables in your containers.

If a container already consumes a Secret in an environment variable, a Secret update will not be seen by the container unless it is restarted. There are third party solutions for triggering restarts when secrets change.

Define a container environment variable with data from a single Secret

- Define an environment variable as a key-value pair in a Secret:

```
kubectl create secret generic backend-user --from-literal=backend
```

- Assign the `backend-username` value defined in the Secret to the `SECRET_USERNAME` environment variable in the Pod specification.

[pods/inject/pod-single-secret-env-variable.yaml](#) 

```
apiVersion: v1
kind: Pod
metadata:
  name: env-single-secret
spec:
  containers:
    - name: envvars-test-container
      image: nginx
      env:
        - name: SECRET_USERNAME
          valueFrom:
            secretKeyRef:
              name: backend-user
              key: backend-username
```

- Create the Pod:

```
kubectl create -f https://k8s.io/examples/pods/inject/pod-single-
```

- In your shell, display the content of `SECRET_USERNAME` container environment variable.

```
kubectl exec -i -t env-single-secret -- /bin/sh -c 'echo $SECRET_
```

The output is similar to:

```
backend-admin
```

Define container environment variables with data from multiple Secrets

- As with the previous example, create the Secrets first.

```
kubectl create secret generic backend-user --from-literal=backend
kubectl create secret generic db-user --from-literal=db-username=
```

- Define the environment variables in the Pod specification.

[pods/inject/pod-multiple-secret-env-variable.yaml](#) 

```
apiVersion: v1
kind: Pod
metadata:
  name: envvars-multiple-secrets
spec:
  containers:
    - name: envvars-test-container
      image: nginx
      env:
        - name: BACKEND_USERNAME
          valueFrom:
            secretKeyRef:
              name: backend-user
              key: backend-username
        - name: DB_USERNAME
          valueFrom:
            secretKeyRef:
              name: db-user
              key: db-username
```

- Create the Pod:

```
kubectl create -f https://k8s.io/examples/pods/inject/pod-multipl
```

- In your shell, display the container environment variables.

```
kubectl exec -i -t envvars-multiple-secrets -- /bin/sh -c 'env |
```

The output is similar to:

```
DB_USERNAME=db-admin
BACKEND_USERNAME=backend-admin
```

Configure all key-value pairs in a Secret as container environment variables

Note: This functionality is available in Kubernetes v1.6 and later.

- Create a Secret containing multiple key-value pairs

```
kubectl create secret generic test-secret --from-literal=username
```

- Use envFrom to define all of the Secret's data as container environment variables. The key from the Secret becomes the environment variable name in the Pod.

[pods/inject/pod-secret-envFrom.yaml](#) 

```
apiVersion: v1
kind: Pod
metadata:
  name: envfrom-secret
spec:
  containers:
    - name: envvars-test-container
      image: nginx
      envFrom:
        - secretRef:
            name: test-secret
```

- Create the Pod:

```
kubectl create -f https://k8s.io/examples/pods/inject/pod-secret-
```

- In your shell, display `username` and `password` container environment variables.

```
kubectl exec -i -t envfrom-secret -- /bin/sh -c 'echo "username:
```

The output is similar to:

```
username: my-app
password: 39528$vdg7Jb
```

Example: Provide prod/test credentials to Pods using Secrets

This example illustrates a Pod which consumes a secret containing production credentials and another Pod which consumes a secret with test environment credentials.

1. Create a secret for prod environment credentials:

```
kubectl create secret generic prod-db-secret --from-literal=username=produser&password=prodpass
```

The output is similar to:

```
secret "prod-db-secret" created
```

2. Create a secret for test environment credentials.

```
kubectl create secret generic test-db-secret --from-literal=username=testuser&password=testpass
```

The output is similar to:

```
secret "test-db-secret" created
```

Note:

Special characters such as \$, \ , * , = , and ! will be interpreted by your [shell](#) and require escaping.

In most shells, the easiest way to escape the password is to surround it with single quotes ('). For example, if your actual password is S!B*d\$zDsb= , you should execute the command as follows:

```
kubectl create secret generic dev-db-secret --from-literal=us
```

You do not need to escape special characters in passwords from files (--from-file).

3. Create the Pod manifests:

```
cat <<EOF > pod.yaml
apiVersion: v1
kind: List
items:
- kind: Pod
  apiVersion: v1
  metadata:
    name: prod-db-client-pod
    labels:
      name: prod-db-client
  spec:
    volumes:
    - name: secret-volume
      secret:
        secretName: prod-db-secret
    containers:
    - name: db-client-container
      image: myClientImage
      volumeMounts:
      - name: secret-volume
        readOnly: true
        mountPath: "/etc/secret-volume"
- kind: Pod
  apiVersion: v1
  metadata:
    name: test-db-client-pod
    labels:
      name: test-db-client
  spec:
    volumes:
    - name: secret-volume
      secret:
        secretName: test-db-secret
    containers:
    - name: db-client-container
      image: myClientImage
      volumeMounts:
      - name: secret-volume
        readOnly: true
        mountPath: "/etc/secret-volume"
EOF
```

Note: How the specs for the two Pods differ only in one field; this facilitates creating Pods with different capabilities from a common Pod template.

4. Apply all those objects on the API server by running:

```
kubectl create -f pod.yaml
```

Both containers will have the following files present on their filesystems with the values for each container's environment:

```
/etc/secret-volume/username  
/etc/secret-volume/password
```

You could further simplify the base Pod specification by using two service accounts:

1. prod-user with the prod-db-secret
2. test-user with the test-db-secret

The Pod specification is shortened to:

```
apiVersion: v1  
kind: Pod  
metadata:  
  name: prod-db-client-pod  
  labels:  
    name: prod-db-client  
spec:  
  serviceAccount: prod-db-client  
  containers:  
  - name: db-client-container  
    image: myClientImage
```

References

- [Secret](#)
- [Volume](#)
- [Pod](#)

What's next

- Learn more about [Secrets](#).
- Learn about [Volumes](#).

8 - Run Applications

Run and manage both stateless and stateful applications.

8.1 - Run a Stateless Application Using a Deployment

This page shows how to run an application using a Kubernetes Deployment object.

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. It is recommended to run this tutorial on a cluster with at least two nodes that are not acting as control plane hosts. If you do not already have a cluster, you can create one by using [minikube](#) or you can use one of these Kubernetes playgrounds:

- [Killercoda](#)
- [Play with Kubernetes](#)

Your Kubernetes server must be at or later than version v1.9. To check the version, enter `kubectl version`.

Creating and exploring an nginx deployment

You can run an application by creating a Kubernetes Deployment object, and you can describe a Deployment in a YAML file. For example, this YAML file describes a Deployment that runs the nginx:1.14.2 Docker image:

[application/deployment.yaml](#) 

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-deployment
spec:
  selector:
    matchLabels:
      app: nginx
  replicas: 2 # tells deployment to run 2 pods matching the template
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
        - name: nginx
          image: nginx:1.14.2
          ports:
            - containerPort: 80
```

1. Create a Deployment based on the YAML file:

```
kubectl apply -f https://k8s.io/examples/application/deployment.y
```

2. Display information about the Deployment:

```
kubectl describe deployment nginx-deployment
```

The output is similar to this:

```
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:1.14.2
      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
nginx-deployment-1771418926-7o5ns	1/1	Running	0
nginx-deployment-1771418926-r18az	1/1	Running	0

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 1.16.1.

[application/deployment-update.yaml](#) 

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-deployment
spec:
  selector:
    matchLabels:
      app: nginx
  replicas: 2
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
        - name: nginx
          image: nginx:1.16.1 # Update the version of nginx from 1.14.2
          ports:
            - containerPort: 80
```

1. Apply the new YAML file:

```
kubectl apply -f https://k8s.io/examples/application/deployment-u
```

2. Watch the deployment create pods with new names and delete the old pods:

```
kubectl get pods -l app=nginx
```

Scaling the application by increasing the replica count

You can increase the number of Pods in your Deployment by applying a new YAML file. This YAML file sets `replicas` to 4, which specifies that the Deployment should have four Pods:

[application/deployment-scale.yaml](#) 

```
apiVersion: apps/v1
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:1.16.1
          ports:
            - containerPort: 80
```

1. Apply the new YAML file:

```
kubectl apply -f https://k8s.io/examples/application/deployment-s
```

2. Verify that the Deployment has four Pods:

```
kubectl get pods -l app=nginx
```

The output is similar to this:

NAME	READY	STATUS	RESTARTS
nginx-deployment-148880595-4zdqq	1/1	Running	0
nginx-deployment-148880595-6zgi1	1/1	Running	0
nginx-deployment-148880595-fxcez	1/1	Running	0
nginx-deployment-148880595-rwovn	1/1	Running	0

Deleting a deployment

Delete the deployment by name:

```
kubectl delete deployment nginx-deployment
```

ReplicationControllers -- the Old Way

The preferred way to create a replicated application is to use a Deployment, which in turn uses a ReplicaSet. Before the Deployment and ReplicaSet were added to Kubernetes, replicated applications were configured using a [ReplicationController](#).

What's next

- Learn more about [Deployment objects](#).

8.2 - 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

- 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. It is recommended to run this tutorial on a cluster with at least two nodes that are not acting as control plane hosts. If you do not already have a cluster, you can create one by using [minikube](#) or you can use one of these Kubernetes playgrounds:
 - [Killercoda](#)
 - [Play with Kubernetes](#)

To check the version, enter `kubectl version`.

- You need to either have a [dynamic PersistentVolume provisioner](#) with a default [StorageClass](#), or [statically provision PersistentVolumes](#) yourself to satisfy the [PersistentVolumeClaims](#) used here.

Deploy MySQL

You can run a stateful application by creating a Kubernetes Deployment and connecting it to an existing PersistentVolume using a PersistentVolumeClaim. For example, this YAML file describes a Deployment that runs MySQL and references the PersistentVolumeClaim. The file defines a volume mount for `/var/lib/mysql`, and then creates a PersistentVolumeClaim that looks for a 20G volume. This claim is satisfied by any existing volume that meets the requirements, or by a dynamic provisioner.

Note: The password is defined in the config yaml, and this is insecure. See [Kubernetes Secrets](#) for a secure solution.

[application/mysql/mysql-deployment.yaml](#) 

```
apiVersion: v1
kind: Service
metadata:
  name: mysql
spec:
  ports:
  - port: 3306
  selector:
    app: mysql
  clusterIP: None
---
apiVersion: apps/v1
kind: Deployment
metadata:
  name: mysql
spec:
  selector:
    matchLabels:
      app: mysql
  strategy:
    type: Recreate
  template:
    metadata:
      labels:
        app: mysql
    spec:
      containers:
      - image: mysql:5.6
        name: mysql
        env:
          # Use secret in real usage
          - name: MYSQL_ROOT_PASSWORD
            value: password
        ports:
        - containerPort: 3306
          name: mysql
      volumeMounts:
      - name: mysql-persistent-storage
        mountPath: /var/lib/mysql
    volumes:
    - name: mysql-persistent-storage
  persistentVolumeClaim:
    claimName: mysql-pv-claim
```

[application/mysql/mysql-pv.yaml](#) 

```
apiVersion: v1
kind: PersistentVolume
metadata:
  name: mysql-pv-volume
  labels:
    type: local
spec:
  storageClassName: manual
  capacity:
    storage: 20Gi
  accessModes:
    - ReadWriteOnce
  hostPath:
    path: "/mnt/data"
---
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: mysql-pv-claim
spec:
  storageClassName: manual
  accessModes:
    - ReadWriteOnce
  resources:
    requests:
      storage: 20Gi
```

1. Deploy the PV and PVC of the YAML file:

```
kubectl apply -f https://k8s.io/examples/application/mysql/mysql-
```

2. Deploy the contents of the YAML file:

```
kubectl apply -f https://k8s.io/examples/application/mysql/mysql-
```

3. Display information about the Deployment:

```
kubectl describe deployment mysql
```

The output is similar to this:

```

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
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)
      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                    SubobjectURL
    -----  -----  -----  ----- 
    33s       33s       1      {deployment-controller } 

```

4. List the pods created by the Deployment:

```
kubectl get pods -l app=mysql
```

The output is similar to this:

NAME	READY	STATUS	RESTARTS	AGE
mysql-63082529-2z3ki	1/1	Running	0	3m

5. Inspect the PersistentVolumeClaim:

```
kubectl describe pvc mysql-pv-claim
```

The output is similar to this:

```
Name: mysql-pv-claim
Namespace: default
StorageClass:
Status: Bound
Volume: mysql-pv-volume
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 --
```

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, stat
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](#)

8.3 - Run a Replicated Stateful Application

This page shows how to run a replicated stateful application using a [StatefulSet](#). This application is a replicated MySQL database. The example topology has a single primary server and multiple replicas, using asynchronous row-based replication.

Note: This is not a production configuration. MySQL settings remain on insecure defaults to keep the focus on general patterns for running stateful applications in Kubernetes.

Before you begin

- You need to have a Kubernetes cluster, and the `kubectl` command-line tool must be configured to communicate with your cluster. It is recommended to run this tutorial on a cluster with at least two nodes that are not acting as control plane hosts. If you do not already have a cluster, you can create one by using [minikube](#) or you can use one of these Kubernetes playgrounds:
 - [Killercoda](#)
 - [Play with Kubernetes](#)
- 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.
- You are using the default namespace or another namespace that does not contain any conflicting objects.

Objectives

- Deploy a replicated MySQL topology with a StatefulSet.
- Send MySQL client traffic.
- Observe resistance to downtime.
- Scale the StatefulSet up and down.

Deploy MySQL

The example MySQL deployment consists of a ConfigMap, two Services, and a StatefulSet.

Create a ConfigMap

Create the ConfigMap from the following YAML configuration file:

[application/mysql/mysql-configmap.yaml](#) 

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: mysql
  labels:
    app: mysql
    app.kubernetes.io/name: mysql
data:
  primary.cnf: |
    # Apply this config only on the primary.
    [mysqlld]
    Log-bin
  replica.cnf: |
    # Apply this config only on replicas.
    [mysqlld]
    super-read-only
```

```
kubectl apply -f https://k8s.io/examples/application/mysql/mysql-configmap.yaml
```

This ConfigMap provides `my.cnf` overrides that let you independently control configuration on the primary MySQL server and its replicas. In this case, you want the primary server to be able to serve replication logs to replicas and you want replicas 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.

Create Services

Create the Services from the following YAML configuration file:

[application/mysql/mysql-services.yaml](#) 

```
# Headless service for stable DNS entries of StatefulSet members.
apiVersion: v1
kind: Service
metadata:
  name: mysql
  labels:
    app: mysql
    app.kubernetes.io/name: 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 primary: mysql-0.mysql.
apiVersion: v1
kind: Service
metadata:
  name: mysql-read
  labels:
    app: mysql
    app.kubernetes.io/name: mysql
    readonly: "true"
spec:
  ports:
  - name: mysql
    port: 3306
  selector:
    app: mysql
```

```
kubectl apply -f https://k8s.io/examples/application/mysql/mysql-service.yaml
```

The headless Service provides a home for the DNS entries that the StatefulSet controllers 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 primary MySQL server and all replicas.

Note that only read queries can use the load-balanced client Service. Because there is only one primary MySQL server, clients should connect directly to the primary MySQL Pod (through its DNS entry within the headless Service) to execute writes.

Create the StatefulSet

Finally, create the StatefulSet from the following YAML configuration file:

[application/mysql/mysql-statefulset.yaml](#) 

```
apiVersion: apps/v1
kind: StatefulSet
metadata:
  name: mysql
spec:
  selector:
    matchLabels:
      app: mysql
      app.kubernetes.io/name: mysql
  serviceName: mysql
  replicas: 3
  template:
    metadata:
      labels:
        app: mysql
        app.kubernetes.io/name: 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=${BASH_REMATCH[1]}
              echo [mysqlId] > /mnt/conf.d/server-id.cnf
              # Add an offset to avoid reserved server-id=0 value.
              echo server-id=$((100 + $ordinal)) >> /mnt/conf.d/server-id.
              # Copy appropriate conf.d files from config-map to emptyDir.
              if [[ $ordinal -eq 0 ]]; then
                cp /mnt/config-map/primary.cnf /mnt/conf.d/
              else
                cp /mnt/config-map/replica.cnf /mnt/conf.d/
              fi
      volumeMounts:
        - name: conf
          mountPath: /mnt/conf.d
        - name: config-map
          mountPath: /mnt/config-map
      containers:
        - name: clone-mysql
          image: gcr.io/google-samples/xtrabackup:1.0
          command:
            - bash
            - "-c"
            - |
              set -ex
              # Skip the clone if data already exists.
              [[ -d /var/lib/mysql/mysql ]] && exit 0
              # Skip the clone on primary (ordinal index 0).
              [[ `hostname` =~ -([0-9]+)$ ]] || exit 1
              ordinal=${BASH_REMATCH[1]}
              [[ $ordinal -eq 0 ]] && exit 0
              # Clone data from previous peer.
              ncat --recv-only mysql-$((ordinal-1)).mysql 3307 | xbstream
              # Prepare the backup.
              xtrabackup --prepare --target-dir=/var/lib/mysql
      volumeMounts:
        - name: data
          mountPath: /var/lib/mysql
          subPath: mysql
        - name: conf
          mountPath: /etc/mysql/conf.d
  containers:
```

```
- name: mysql
  image: mysql:5.7
  env:
    - name: MYSQL_ALLOW_EMPTY_PASSWORD
      value: "1"
  ports:
    - name: mysql
      containerPort: 3306
  volumeMounts:
    - name: data
      mountPath: /var/lib/mysql
      subPath: mysql
    - name: conf
      mountPath: /etc/mysql/conf.d
  resources:
    requests:
      cpu: 500m
      memory: 1Gi
  livenessProbe:
    exec:
      command: ["mysqladmin", "ping"]
    initialDelaySeconds: 30
    periodSeconds: 10
    timeoutSeconds: 5
  readinessProbe:
    exec:
      # Check we can execute queries over TCP (skip-networking is disabled)
      command: ["mysql", "-h", "127.0.0.1", "-e", "SELECT 1"]
    initialDelaySeconds: 5
    periodSeconds: 2
    timeoutSeconds: 1
- name: xtrabackup
  image: gcr.io/google-samples/xtrabackup:1.0
  ports:
    - name: xtrabackup
      containerPort: 3307
  command:
    - bash
    - "-c"
    - |
      set -ex
      cd /var/lib/mysql

      # Determine binlog position of cloned data, if any.
      if [[ -f xtrabackup_slave_info && "x$(<xtrabackup_slave_info
        # XtraBackup already generated a partial "CHANGE MASTER TO
        # because we're cloning from an existing replica. (Need to
        cat xtrabackup_slave_info | sed -E 's/;$/;/g' > change_master_to
        # Ignore xtrabackup_binlog_info in this case (it's useless
        rm -f xtrabackup_slave_info xtrabackup_binlog_info
      elif [[ -f xtrabackup_binlog_info ]]; then
        # We're cloning directly from primary. Parse binlog position
        [[ `cat xtrabackup_binlog_info` =~ ^(.*)[:space:]+(.*)?]
        rm -f xtrabackup_binlog_info xtrabackup_slave_info
        echo "CHANGE MASTER TO MASTER_LOG_FILE='${BASH_REMATCH[1]}'
          MASTER_LOG_POS=${BASH_REMATCH[2]} > change_master_to
      fi

      # Check if we need to complete a clone by starting replication
      if [[ -f change_master_to.sql.in ]]; then
        echo "Waiting for mysqld to be ready (accepting connections)
        until mysql -h 127.0.0.1 -e "SELECT 1"; do sleep 1; done

        echo "Initializing replication from clone position"
        mysql -h 127.0.0.1 \
          -e "$(<change_master_to.sql.in), \
            MASTER_HOST='mysql-0.mysql', \
```

```

MASTER_USER='root', \
MASTER_PASSWORD='', \
MASTER_CONNECT_RETRY=10; \
START SLAVE;" // exit 1

# In case of container restart, attempt this at-most-once.
mv change_master_to.sql.in change_master_to.sql.orig
fi

# Start a server to send backups when requested by peers.
exec ncat --listen --keep-open --send-only --max-conns=1 3306
"xtrabackup --backup --slave-info --stream=xbstream --host
volumeMounts:
- name: data
  mountPath: /var/lib/mysql
  subPath: mysql
- name: conf
  mountPath: /etc/mysql/conf.d
resources:
  requests:
    cpu: 100m
    memory: 100Mi
volumes:
- name: conf
  emptyDir: {}
- name: config-map
  configMap:
    name: mysql
volumeClaimTemplates:
- metadata:
    name: data
  spec:
    accessModes: ["ReadWriteOnce"]
  resources:
    requests:
      storage: 10Gi

```

```
kubectl apply -f https://k8s.io/examples/application/mysql/mysql-statefulset.yaml
```

You can watch the startup progress by running:

```
kubectl get pods -l app=mysql --watch
```

After a while, you should see all 3 Pods become `Running`:

NAME	READY	STATUS	RESTARTS	AGE
mysql-0	2/2	Running	0	2m
mysql-1	2/2	Running	0	1m
mysql-2	2/2	Running	0	1m

Press **Ctrl+C** to cancel the watch.

Note: If you don't see any progress, make sure you have a dynamic PersistentVolume provisioner enabled, as mentioned in the [prerequisites](#).

This manifest uses a variety of techniques for managing stateful Pods as part of a StatefulSet. The next section highlights some of these techniques to explain what happens as the StatefulSet creates Pods.

Understanding stateful Pod initialization

The StatefulSet controller starts Pods one at a time, in order by their ordinal index. It waits until each Pod reports being Ready before starting the next one.

In addition, the controller assigns each Pod a unique, stable name of the form `<statefulset-name>-<ordinal-index>`, which results in Pods named `mysql-0`, `mysql-1`, and `mysql-2`.

The Pod template in the above StatefulSet manifest takes advantage of these properties to perform orderly startup of MySQL replication.

Generating configuration

Before starting any of the containers in the Pod spec, the Pod first runs any [init containers](#) in the order defined.

The first init container, named `init-mysql`, generates special MySQL config files based on the ordinal index.

The script determines its own ordinal index by extracting it from the end of the Pod name, which is returned by the `hostname` command. Then it saves the ordinal (with a numeric offset to avoid reserved values) into a file called `server-id.cnf` in the MySQL `conf.d` directory. This translates the unique, stable identity provided by the StatefulSet into the domain of MySQL server IDs, which require the same properties.

The script in the `init-mysql` container also applies either `primary.cnf` or `replica.cnf` from the ConfigMap by copying the contents into `conf.d`. Because the example topology consists of a single primary MySQL server and any number of replicas, the script assigns ordinal `0` to be the primary server, and everyone else to be replicas. Combined with the StatefulSet controller's [deployment order guarantee](#), this ensures the primary MySQL server is Ready before creating replicas, so they can begin replicating.

Cloning existing data

In general, when a new Pod joins the set as a replica, it must assume the primary MySQL server 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 replica 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 primary server.

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 primary MySQL server, 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 replica. 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 replica begins replication, it remembers its primary MySQL server and reconnects automatically if the server restarts or the connection dies. Also, because replicas look for the primary server at its stable DNS name (`mysql-0.mysql`), they automatically find the primary server 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 primary MySQL server (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,
+-----+
| 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'; 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.

Simulate Pod and Node failure

To demonstrate the increased availability of reading from the pool of replicas 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	kuberr

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.

Caution: Draining a Node can impact other workloads and applications running on the same node. Only perform the following step in a test cluster.

```
# See above advice about impact on other workloads
kubectl drain <node-name> --force --delete-emptydir-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_NAME
mysql-2	2/2	Terminating	0	15m	10.244.1.56	kube-node-1
[...]						
mysql-2	0/2	Pending	0	0s	<none>	kube-node-1
mysql-2	0/2	Init:0/2	0	0s	<none>	kube-node-1
mysql-2	0/2	Init:1/2	0	20s	10.244.5.32	kube-node-1
mysql-2	0/2	PodInitializing	0	21s	10.244.5.32	kube-node-1
mysql-2	1/2	Running	0	22s	10.244.5.32	kube-node-1
mysql-2	2/2	Running	0	30s	10.244.5.32	kube-node-1

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 replicas

When you use MySQL replication, you can scale your read query capacity by adding replicas. For a StatefulSet, you can achieve 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,
+-----+
| message |
+-----+
| hello   |
+-----+
pod "mysql-client" deleted
```

Scaling back down is also seamless:

```
kubectl scale statefulset mysql --replicas=3
```

Note:

Although 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
data-mysql-0	Bound	pvc-8acbf5dc-b103-11e6-93fa-42010a800002	100Gi
data-mysql-1	Bound	pvc-8ad39820-b103-11e6-93fa-42010a800002	100Gi
data-mysql-2	Bound	pvc-8ad69a6d-b103-11e6-93fa-42010a800002	100Gi
data-mysql-3	Bound	pvc-50043c45-b1c5-11e6-93fa-42010a800002	100Gi
data-mysql-4	Bound	pvc-500a9957-b1c5-11e6-93fa-42010a800002	100Gi

If you don't intend to reuse the extra PVCs, you can delete them:

```
kubectl delete pvc data-mysql-3  
kubectl delete pvc data-mysql-4
```

Cleaning up

1. Cancel the `SELECT @@server_id` loop by pressing **Ctrl+C** in its terminal, or running the following from another terminal:

```
kubectl delete pod mysql-client-loop --now
```

2. Delete the StatefulSet. This also begins terminating the Pods.

```
kubectl delete statefulset mysql
```

3. Verify that the Pods disappear. They might take some time to finish terminating.

```
kubectl get pods -l app=mysql
```

You'll know the Pods have terminated when the above returns:

```
No resources found.
```

4. Delete the ConfigMap, Services, and PersistentVolumeClaims.

```
kubectl delete configmap,service,pvc -l app=mysql
```

5. If you manually provisioned PersistentVolumes, you also need to manually delete them, as well as release the underlying resources. If you used a dynamic provisioner, it automatically deletes the PersistentVolumes when it sees that you deleted the PersistentVolumeClaims. Some dynamic provisioners (such as those for EBS and PD) also release the underlying resources upon deleting the PersistentVolumes.

What's next

- Learn more about [scaling a StatefulSet](#).
- Learn more about [debugging a StatefulSet](#).
- Learn more about [deleting a StatefulSet](#).
- Learn more about [force deleting StatefulSet Pods](#).
- Look in the [Helm Charts repository](#) for other stateful application examples.

8.4 - Scale a StatefulSet

This task shows how to scale a StatefulSet. Scaling a StatefulSet refers to increasing or decreasing the number of replicas.

Before you begin

- StatefulSets are only available in Kubernetes version 1.5 or later. To check your version of Kubernetes, run `kubectl version`.
- Not all stateful applications scale nicely. If you are unsure about whether to scale your StatefulSets, see [StatefulSet concepts](#) or [StatefulSet tutorial](#) for further information.
- You should perform scaling only when you are confident that your stateful application cluster is completely healthy.

Scaling StatefulSets

Use kubectl to scale StatefulSets

First, find the StatefulSet you want to scale.

```
kubectl get statefulsets <stateful-set-name>
```

Change the number of replicas of your StatefulSet:

```
kubectl scale statefulsets <stateful-set-name> --replicas=<new-replica>
```

Make in-place updates on your StatefulSets

Alternatively, you can do [in-place updates](#) on your StatefulSets.

If your StatefulSet was initially created with `kubectl apply`, update `.spec.replicas` of the StatefulSet manifests, and then do a `kubectl apply`:

```
kubectl apply -f <stateful-set-file-updated>
```

Otherwise, edit that field with `kubectl edit`:

```
kubectl edit statefulsets <stateful-set-name>
```

Or use `kubectl patch`:

```
kubectl patch statefulsets <stateful-set-name> -p '{"spec":{"replicas":<new-replica>}}
```

Troubleshooting

Scaling down does not work right

You cannot scale down a StatefulSet when any of the stateful Pods it manages is unhealthy. Scaling down only takes place after those stateful Pods become running and ready.

If `spec.replicas > 1`, Kubernetes cannot determine the reason for an unhealthy Pod. It might be the result of a permanent fault or of a transient fault. A transient fault can be caused by a restart required by upgrading or maintenance.

If the Pod is unhealthy due to a permanent fault, scaling without correcting the fault may lead to a state where the StatefulSet membership drops below a certain minimum number of replicas that are needed to function correctly. This may cause your StatefulSet to become unavailable.

If the Pod is unhealthy due to a transient fault and the Pod might become available again, the transient error may interfere with your scale-up or scale-down operation. Some distributed databases have issues when nodes join and leave at the same time. It is better to reason about scaling operations at the application level in these cases, and perform scaling only when you are sure that your stateful application cluster is completely healthy.

What's next

- Learn more about [deleting a StatefulSet](#).

8.5 - Delete a StatefulSet

This task shows you how to delete a StatefulSet.

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>
```

When deleting a StatefulSet through `kubectl`, the StatefulSet scales down to 0. All Pods that are part of this workload are also deleted. If you want to delete only the StatefulSet and not the Pods, use `--cascade=orphan`. For example:

```
kubectl delete -f <file.yaml> --cascade=orphan
```

By passing `--cascade=orphan` 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.kubernetes.io/name=MyApp`, you can then delete them as follows:

```
kubectl delete pods -l app.kubernetes.io/name=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 terminated 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 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.kubernetes.io/name=MyApp
sleep $grace
kubectl delete pvc -l app.kubernetes.io/name=MyApp
```

In the example above, the Pods have the label `app.kubernetes.io/name=MyApp`; substitute your own label as appropriate.

Force deletion of StatefulSet pods

If you find that some pods in your StatefulSet are stuck in the 'Terminating' or 'Unknown' states for an extended period of time, you may need to manually intervene to forcefully delete the pods from the apiserver. This is a potentially dangerous task. Refer to [Force Delete StatefulSet Pods](#) for details.

What's next

Learn more about [force deleting StatefulSet Pods](#).

8.6 - 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

- 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.

A Pod is not deleted automatically when 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
```

If even after these commands the pod is stuck on `Unknown` state, use the following command to remove the pod from the cluster:

```
kubectl patch pod <pod> -p '{"metadata":{"finalizers":null}}'
```

Always perform force deletion of StatefulSet Pods carefully and with complete knowledge of the risks involved.

What's next

Learn more about [debugging a StatefulSet](#).

8.7 - Horizontal Pod Autoscaling

In Kubernetes, a *HorizontalPodAutoscaler* automatically updates a workload resource (such as a [Deployment](#) or [StatefulSet](#)), with the aim of automatically scaling the workload to match demand.

Horizontal scaling means that the response to increased load is to deploy more [Pods](#). This is different from *vertical* scaling, which for Kubernetes would mean assigning more resources (for example: memory or CPU) to the Pods that are already running for the workload.

If the load decreases, and the number of Pods is above the configured minimum, the *HorizontalPodAutoscaler* instructs the workload resource (the Deployment, StatefulSet, or other similar resource) to scale back down.

Horizontal pod autoscaling does not apply to objects that can't be scaled (for example: a [DaemonSet](#).)

The *HorizontalPodAutoscaler* is implemented as a Kubernetes API resource and a [controller](#). The resource determines the behavior of the controller. The horizontal pod autoscaling controller, running within the Kubernetes control plane, periodically adjusts the desired scale of its target (for example, a Deployment) to match observed metrics such as average CPU utilization, average memory utilization, or any other custom metric you specify.

There is [walkthrough example](#) of using horizontal pod autoscaling.

How does a *HorizontalPodAutoscaler* work?

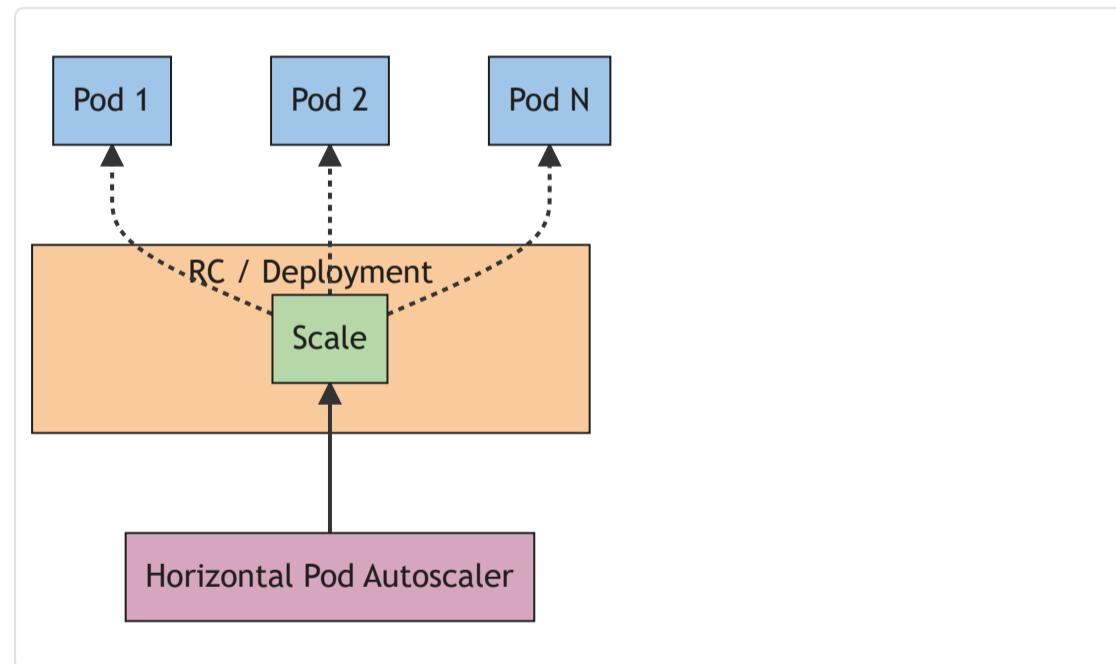


Figure 1. *HorizontalPodAutoscaler* controls the scale of a Deployment and its ReplicaSet

Kubernetes implements horizontal pod autoscaling as a control loop that runs intermittently (it is not a continuous process). The interval is set by the `--horizontal-pod-autoscaler-sync-period` parameter to the [kube-controller-manager](#) (and the default interval is 15 seconds).

Once during each period, the controller manager queries the resource utilization against the metrics specified in each *HorizontalPodAutoscaler* definition. The controller manager finds the target resource defined by the `scaleTargetRef`, then selects the pods based on the target resource's

.spec.selector labels, and obtains the metrics from either the resource metrics API (for per-pod resource metrics), or the custom metrics API (for all other metrics).

- For per-pod resource metrics (like CPU), the controller fetches the metrics from the resource metrics API for each Pod targeted by the HorizontalPodAutoscaler. Then, if a target utilization value is set, the controller calculates the utilization value as a percentage of the equivalent [resource request](#) on the containers in each Pod. If a target raw value is set, the raw metric values are used directly. The controller then takes the mean of the utilization or the raw value (depending on the type of target specified) across all targeted Pods, and produces a ratio used to scale the number of desired replicas.

Please note that if some of the Pod's containers do not have the relevant resource request set, CPU utilization for the Pod will not be defined and the autoscaler will not take any action for that metric. See the [algorithm details](#) section below for more information about how the autoscaling algorithm works.

- For per-pod custom metrics, the controller functions similarly to per-pod resource metrics, except that it works with raw values, not utilization values.
- For object metrics and external metrics, a single metric is fetched, which describes the object in question. This metric is compared to the target value, to produce a ratio as above. In the `autoscaling/v2` API version, this value can optionally be divided by the number of Pods before the comparison is made.

The common use for HorizontalPodAutoscaler is to configure it to fetch metrics from [aggregated APIs](#) (`metrics.k8s.io`, `custom.metrics.k8s.io`, or `external.metrics.k8s.io`). The `metrics.k8s.io` API is usually provided by an add-on named Metrics Server, which needs to be launched separately. For more information about resource metrics, see [Metrics Server](#).

[Support for metrics APIs](#) explains the stability guarantees and support status for these different APIs.

The HorizontalPodAutoscaler controller accesses corresponding workload resources that support scaling (such as Deployments and StatefulSet). These resources each have a subresource named `scale`, an interface that allows you to dynamically set the number of replicas and examine each of their current states. For general information about subresources in the Kubernetes API, see [Kubernetes API Concepts](#).

Algorithm details

From the most basic perspective, the HorizontalPodAutoscaler controller operates on the ratio between desired metric value and current metric value:

```
desiredReplicas = ceil[currentReplicas * ( currentMetricValue / desired
```

For example, if the current metric value is `200m`, and the desired value is `100m`, the number of replicas will be doubled, since $200.0 / 100.0 == 2.0$. If the current value is instead `50m`, you'll halve the number of replicas, since $50.0 / 100.0 == 0.5$. The control plane skips any scaling action if the ratio is sufficiently close to 1.0 (within a globally-configurable tolerance, 0.1 by default).

When a `targetAverageValue` or `targetAverageUtilization` is specified, the `currentMetricValue` is computed by taking the average of the given metric across all Pods in the HorizontalPodAutoscaler's scale target.

Before checking the tolerance and deciding on the final values, the control plane also considers whether any metrics are missing, and how many Pods are [Ready](#). All Pods with a deletion timestamp set (objects with a deletion timestamp are in the process of being shut down / removed) are ignored, and all failed Pods are discarded.

If a particular Pod is missing metrics, it is set aside for later; Pods with missing metrics will be used to adjust the final scaling amount.

When scaling on CPU, if any pod has yet to become ready (it's still initializing, or possibly is unhealthy) or the most recent metric point for the pod was before it became ready, that pod is set aside as well.

Due to technical constraints, the HorizontalPodAutoscaler controller cannot exactly determine the first time a pod becomes ready when determining whether to set aside certain CPU metrics. Instead, it considers a Pod "not yet ready" if it's unready and transitioned to ready within a short, configurable window of time since it started. This value is configured with the `--horizontal-pod-autoscaler-initial-readiness-delay` flag, and its default is 30 seconds. Once a pod has become ready, it considers any transition to ready to be the first if it occurred within a longer, configurable time since it started. This value is configured with the `--horizontal-pod-autoscaler-cpu-initialization-period` flag, and its default is 5 minutes.

The `currentMetricValue / desiredMetricValue` base scale ratio is then calculated using the remaining pods not set aside or discarded from above.

If there were any missing metrics, the control plane recomputes the average more conservatively, assuming those pods were consuming 100% of the desired value in case of a scale down, and 0% in case of a scale up. This dampens the magnitude of any potential scale.

Furthermore, if any not-yet-ready pods were present, and the workload would have scaled up without factoring in missing metrics or not-yet-ready pods, the controller conservatively assumes that the not-yet-ready pods are consuming 0% of the desired metric, further dampening the magnitude of a scale up.

After factoring in the not-yet-ready pods and missing metrics, the controller recalculates the usage ratio. If the new ratio reverses the scale direction, or is within the tolerance, the controller doesn't take any scaling action. In other cases, the new ratio is used to decide any change to the number of Pods.

Note that the *original* value for the average utilization is reported back via the HorizontalPodAutoscaler status, without factoring in the not-yet-ready pods or missing metrics, even when the new usage ratio is used.

If multiple metrics are specified in a HorizontalPodAutoscaler, this calculation is done for each metric, and then the largest of the desired replica counts is chosen. If any of these metrics cannot be converted into a desired replica count (e.g. due to an error fetching the metrics from the metrics APIs) and a scale down is suggested by the metrics which can be fetched, scaling is skipped. This means that the HPA is still capable of scaling up if one or more metrics give a `desiredReplicas` greater than the current value.

Finally, right before HPA scales the target, the scale recommendation is recorded. The controller considers all recommendations within a configurable window choosing the highest recommendation from within that window. This value can be configured using the `--horizontal-pod-`

`autoscaler-downscale-stabilization` flag, which defaults to 5 minutes. This means that scaledowns will occur gradually, smoothing out the impact of rapidly fluctuating metric values.

API Object

The Horizontal Pod Autoscaler is an API resource in the Kubernetes `autoscaling` API group. The current stable version can be found in the `autoscaling/v2` API version which includes support for scaling on memory and custom metrics. The new fields introduced in `autoscaling/v2` are preserved as annotations when working with `autoscaling/v1`.

When you create a `HorizontalPodAutoscaler` API object, make sure the name specified is a valid [DNS subdomain name](#). More details about the API object can be found at [HorizontalPodAutoscaler Object](#).

Stability of workload scale

When managing the scale of a group of replicas using the `HorizontalPodAutoscaler`, 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*, or *flapping*. It's similar to the concept of *hysteresis* in cybernetics.

Autoscaling during rolling update

Kubernetes lets you perform a rolling update on a Deployment. In that case, the Deployment manages the underlying ReplicaSets for you. When you configure autoscaling for a Deployment, you bind a `HorizontalPodAutoscaler` to a single Deployment. The `HorizontalPodAutoscaler` manages the `replicas` field of the Deployment. The deployment controller is responsible for setting the `replicas` of the underlying ReplicaSets so that they add up to a suitable number during the rollout and also afterwards.

If you perform a rolling update of a StatefulSet that has an autoscaled number of replicas, the StatefulSet directly manages its set of Pods (there is no intermediate resource similar to ReplicaSet).

Support for resource metrics

Any HPA target can be scaled based on the resource usage of the pods in the scaling target. When defining the pod specification the resource requests like `cpu` and `memory` should be specified. This is used to determine the resource utilization and used by the HPA controller to scale the target up or down. To use resource utilization based scaling specify a metric source like this:

```
type: Resource
resource:
  name: cpu
  target:
    type: Utilization
    averageUtilization: 60
```

With this metric the HPA controller will keep the average utilization of the pods in the scaling target at 60%. Utilization is the ratio between the current usage of resource to the requested resources of the pod. See [Algorithm](#) for more details about how the utilization is calculated and averaged.

Note: Since the resource usages of all the containers are summed up the total pod utilization may not accurately represent the individual container resource usage. This could lead to situations where a single container might be running with high usage and the HPA will not scale out because the overall pod usage is still within acceptable limits.

Container resource metrics

FEATURE STATE: [Kubernetes v1.27 \[beta\]](#)

The HorizontalPodAutoscaler API also supports a container metric source where the HPA can track the resource usage of individual containers across a set of Pods, in order to scale the target resource. This lets you configure scaling thresholds for the containers that matter most in a particular Pod. For example, if you have a web application and a logging sidecar, you can scale based on the resource use of the web application, ignoring the sidecar container and its resource use.

If you revise the target resource to have a new Pod specification with a different set of containers, you should revise the HPA spec if that newly added container should also be used for scaling. If the specified container in the metric source is not present or only present in a subset of the pods then those pods are ignored and the recommendation is recalculated. See [Algorithm](#) for more details about the calculation. To use container resources for autoscaling define a metric source as follows:

```
type: ContainerResource
containerResource:
  name: cpu
  container: application
  target:
    type: Utilization
    averageUtilization: 60
```

In the above example the HPA controller scales the target such that the average utilization of the `cpu` in the `application` container of all the pods is 60%.

Note:

If you change the name of a container that a HorizontalPodAutoscaler is tracking, you can make that change in a specific order to ensure scaling remains available and effective whilst the change is being applied. Before you update the resource that defines the container (such as a Deployment), you should update the associated HPA to track both the new and old container names. This way, the HPA is able to calculate a scaling recommendation throughout the update process.

Once you have rolled out the container name change to the workload resource, tidy up by removing the old container name from the HPA specification.

Scaling on custom metrics

FEATURE STATE: [Kubernetes v1.23 \[stable\]](#)

(the `autoscaling/v2beta2` API version previously provided this ability as a beta feature)

Provided that you use the `autoscaling/v2` API version, you can configure a `HorizontalPodAutoscaler` to scale based on a custom metric (that is not built in to Kubernetes or any Kubernetes component). The `HorizontalPodAutoscaler` controller then queries for these custom metrics from the Kubernetes API.

See [Support for metrics APIs](#) for the requirements.

Scaling on multiple metrics

FEATURE STATE: [Kubernetes v1.23 \[stable\]](#)

(the `autoscaling/v2beta2` API version previously provided this ability as a beta feature)

Provided that you use the `autoscaling/v2` API version, you can specify multiple metrics for a `HorizontalPodAutoscaler` to scale on. Then, the `HorizontalPodAutoscaler` controller evaluates each metric, and proposes a new scale based on that metric. The `HorizontalPodAutoscaler` takes the maximum scale recommended for each metric and sets the workload to that size (provided that this isn't larger than the overall maximum that you configured).

Support for metrics APIs

By default, the `HorizontalPodAutoscaler` controller retrieves metrics from a series of APIs. In order for it to access these APIs, cluster administrators must ensure that:

- The [API aggregation layer](#) is enabled.
- The corresponding APIs are registered:
 - For resource metrics, this is the `metrics.k8s.io` [API](#), generally provided by [metrics-server](#). It can be launched as a cluster add-on.
 - For custom metrics, this is the `custom.metrics.k8s.io` [API](#). It's provided by "adapter" API servers provided by metrics solution vendors. Check with your metrics pipeline to see if there is a Kubernetes metrics adapter available.
 - For external metrics, this is the `external.metrics.k8s.io` [API](#). It may be provided by the custom metrics adapters provided above.

For more information on these different metrics paths and how they differ please see the relevant design proposals for [the HPA V2](#), [custom.metrics.k8s.io](#) and [external.metrics.k8s.io](#).

For examples of how to use them see [the walkthrough for using custom metrics](#) and [the walkthrough for using external metrics](#).

Configurable scaling behavior

FEATURE STATE: [Kubernetes v1.23 \[stable\]](#)

(the `autoscaling/v2beta2` API version previously provided this ability as a beta feature)

If you use the `v2` `HorizontalPodAutoscaler` API, you can use the `behavior` field (see the [API reference](#)) to configure separate scale-up and scale-down behaviors. You specify these behaviours by setting `scaleUp` and / or `scaleDown` under the `behavior` field.

You can specify a *stabilization window* that prevents [flapping](#) the replica count for a scaling target. Scaling policies also let you control the rate of change of replicas while scaling.

Scaling policies

One or more scaling policies can be specified in the `behavior` section of the spec. When multiple policies are specified the policy which allows the highest amount of change is the policy which is selected by default. The following example shows this behavior while scaling down:

```
behavior:  
  scaleDown:  
    policies:  
      - type: Pods  
        value: 4  
        periodSeconds: 60  
      - type: Percent  
        value: 10  
        periodSeconds: 60
```

`periodSeconds` indicates the length of time in the past for which the policy must hold true. The maximum value that you can set for `periodSeconds` is 1800 (half an hour). The first policy (*Pods*) allows at most 4 replicas to be scaled down in one minute. The second policy (*Percent*) allows at most 10% of the current replicas to be scaled down in one minute.

Since by default the policy which allows the highest amount of change is selected, the second policy will only be used when the number of pod replicas is more than 40. With 40 or less replicas, the first policy will be applied. For instance if there are 80 replicas and the target has to be scaled down to 10 replicas then during the first step 8 replicas will be reduced. In the next iteration when the number of replicas is 72, 10% of the pods is 7.2 but the number is rounded up to 8. On each loop of the autoscaler controller the number of pods to be change is re-calculated based on the number of current replicas. When the number of replicas falls below 40 the first policy (*Pods*) is applied and 4 replicas will be reduced at a time.

The policy selection can be changed by specifying the `selectPolicy` field for a scaling direction. By setting the value to `Min` which would select the policy which allows the smallest change in the replica count. Setting the value to `Disabled` completely disables scaling in that direction.

Stabilization window

The stabilization window is used to restrict the [flapping](#) of replica count when the metrics used for scaling keep fluctuating. The autoscaling algorithm uses this window to infer a previous desired state and avoid unwanted changes to workload scale.

For example, in the following example snippet, a stabilization window is specified for `scaleDown`.

```
behavior:
  scaleDown:
    stabilizationWindowSeconds: 300
```

When the metrics indicate that the target should be scaled down the algorithm looks into previously computed desired states, and uses the highest value from the specified interval. In the above example, all desired states from the past 5 minutes will be considered.

This approximates a rolling maximum, and avoids having the scaling algorithm frequently remove Pods only to trigger recreating an equivalent Pod just moments later.

Default Behavior

To use the custom scaling not all fields have to be specified. Only values which need to be customized can be specified. These custom values are merged with default values. The default values match the existing behavior in the HPA algorithm.

```
behavior:
  scaleDown:
    stabilizationWindowSeconds: 300
  policies:
    - type: Percent
      value: 100
      periodSeconds: 15
  scaleUp:
    stabilizationWindowSeconds: 0
  policies:
    - type: Percent
      value: 100
      periodSeconds: 15
    - type: Pods
      value: 4
      periodSeconds: 15
  selectPolicy: Max
```

For scaling down the stabilization window is 300 seconds (or the value of the `--horizontal-pod-autoscaler-downscale-stabilization` flag if provided). There is only a single policy for scaling down which allows a 100% of the currently running replicas to be removed which means the scaling target can be scaled down to the minimum allowed replicas. For scaling up there is no stabilization window. When the metrics indicate that the target should be scaled up the target is scaled up immediately. There are 2 policies where 4 pods or a 100% of the currently running replicas may at most be added every 15 seconds till the HPA reaches its steady state.

Example: change downscale stabilization window

To provide a custom downscale stabilization window of 1 minute, the following behavior would be added to the HPA:

```
behavior:
  scaleDown:
    stabilizationWindowSeconds: 60
```

Example: limit scale down rate

To limit the rate at which pods are removed by the HPA to 10% per minute, the following behavior would be added to the HPA:

```
behavior:  
  scaleDown:  
    policies:  
      - type: Percent  
        value: 10  
        periodSeconds: 60
```

To ensure that no more than 5 Pods are removed per minute, you can add a second scale-down policy with a fixed size of 5, and set `selectPolicy` to `minimum`. Setting `selectPolicy` to `Min` means that the autoscaler chooses the policy that affects the smallest number of Pods:

```
behavior:  
  scaleDown:  
    policies:  
      - type: Percent  
        value: 10  
        periodSeconds: 60  
      - type: Pods  
        value: 5  
        periodSeconds: 60  
    selectPolicy: Min
```

Example: disable scale down

The `selectPolicy` value of `Disabled` turns off scaling the given direction. So to prevent downscaling the following policy would be used:

```
behavior:  
  scaleDown:  
    selectPolicy: Disabled
```

Support for HorizontalPodAutoscaler in kubectl

HorizontalPodAutoscaler, like every API resource, is supported in a standard way by `kubectl`. You can create a new autoscaler using `kubectl create` command. You can list autoscalers by `kubectl get hpa` or get detailed description by `kubectl describe hpa`. Finally, you can delete an autoscaler using `kubectl delete hpa`.

In addition, there is a special `kubectl autoscale` command for creating a HorizontalPodAutoscaler object. For instance, executing `kubectl autoscale rs foo --min=2 --max=5 --cpu-percent=80` will create an autoscaler for ReplicaSet `foo`, with target CPU utilization set to 80% and the number of replicas between 2 and 5.

Implicit maintenance-mode deactivation

You can implicitly deactivate the HPA for a target without the need to change the HPA configuration itself. If the target's desired replica count is set to 0, and the HPA's minimum replica count is greater than 0, the HPA stops adjusting the target (and sets the `ScalingActive` Condition on itself to `false`) until you reactivate it by manually adjusting the target's desired replica count or HPA's minimum replica count.

Migrating Deployments and StatefulSets to horizontal autoscaling

When an HPA is enabled, it is recommended that the value of `spec.replicas` of the Deployment and / or StatefulSet be removed from their manifest(s). If this isn't done, any time a change to that object is applied, for example via `kubectl apply -f deployment.yaml`, this will instruct Kubernetes to scale the current number of Pods to the value of the `spec.replicas` key. This may not be desired and could be troublesome when an HPA is active.

Keep in mind that the removal of `spec.replicas` may incur a one-time degradation of Pod counts as the default value of this key is 1 (reference [Deployment Replicas](#)). Upon the update, all Pods except 1 will begin their termination procedures. Any deployment application afterwards will behave as normal and respect a rolling update configuration as desired. You can avoid this degradation by choosing one of the following two methods based on how you are modifying your deployments:

[Client Side Apply \(this is the default\)](#)

[Server Side Apply](#)

1. `kubectl apply edit-last-applied deployment/<deployment_name>`
2. In the editor, remove `spec.replicas`. When you save and exit the editor, `kubectl` applies the update. No changes to Pod counts happen at this step.
3. You can now remove `spec.replicas` from the manifest. If you use source code management, also commit your changes or take whatever other steps for revising the source code are appropriate for how you track updates.
4. From here on out you can run `kubectl apply -f deployment.yaml`

What's next

If you configure autoscaling in your cluster, you may also want to consider running a cluster-level autoscaler such as [Cluster Autoscaler](#).

For more information on HorizontalPodAutoscaler:

- Read a [walkthrough example](#) for horizontal pod autoscaling.
- Read documentation for [kubectl autoscale](#).
- If you would like to write your own custom metrics adapter, check out the [boilerplate](#) to get started.
- Read the [API reference](#) for HorizontalPodAutoscaler.

8.8 - HorizontalPodAutoscaler Walkthrough

A [HorizontalPodAutoscaler](#) (HPA for short) automatically updates a workload resource (such as a [Deployment](#) or [StatefulSet](#)), with the aim of automatically scaling the workload to match demand.

Horizontal scaling means that the response to increased load is to deploy more [Pods](#). This is different from *vertical* scaling, which for Kubernetes would mean assigning more resources (for example: memory or CPU) to the Pods that are already running for the workload.

If the load decreases, and the number of Pods is above the configured minimum, the HorizontalPodAutoscaler instructs the workload resource (the Deployment, StatefulSet, or other similar resource) to scale back down.

This document walks you through an example of enabling HorizontalPodAutoscaler to automatically manage scale for an example web app. This example workload is Apache httpd running some PHP code.

Before you begin

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

- [Killercoda](#)
- [Play with Kubernetes](#)

Your Kubernetes server must be at or later than version 1.23. To check the version, enter `kubectl version`. If you're running an older release of Kubernetes, refer to the version of the documentation for that release (see [available documentation versions](#)).

To follow this walkthrough, you also need to use a cluster that has a [Metrics Server](#) deployed and configured. The Kubernetes Metrics Server collects resource metrics from the [kubelets](#) in your cluster, and exposes those metrics through the [Kubernetes API](#), using an [APIService](#) to add new kinds of resource that represent metric readings.

To learn how to deploy the Metrics Server, see the [metrics-server documentation](#).

Run and expose php-apache server

To demonstrate a HorizontalPodAutoscaler, you will first start a Deployment that runs a container using the `hpa-example` image, and expose it as a Service using the following manifest:

[application/php-apache.yaml](#) 

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: php-apache
spec:
  selector:
    matchLabels:
      run: php-apache
  template:
    metadata:
      labels:
        run: php-apache
    spec:
      containers:
        - name: php-apache
          image: registry.k8s.io/hpa-example
          ports:
            - containerPort: 80
          resources:
            limits:
              cpu: 500m
            requests:
              cpu: 200m
---
apiVersion: v1
kind: Service
metadata:
  name: php-apache
  labels:
    run: php-apache
spec:
  ports:
    - port: 80
  selector:
    run: php-apache
```

To do so, run the following command:

```
kubectl apply -f https://k8s.io/examples/application/php-apache.yaml
```

```
deployment.apps/php-apache created
service/php-apache created
```

Create the HorizontalPodAutoscaler

Now that the server is running, create the autoscaler using `kubectl`. There is `kubectl autoscale` subcommand, part of `kubectl`, that helps you do this.

You will shortly run a command that creates a HorizontalPodAutoscaler that maintains between 1 and 10 replicas of the Pods controlled by the `php-apache` Deployment that you created in the first step of these instructions.

Roughly speaking, the HPA controller will increase and decrease the number of replicas (by updating the Deployment) to maintain an average CPU utilization across all Pods of 50%. The Deployment then updates the ReplicaSet - this is part of how all Deployments work in Kubernetes - and then the ReplicaSet either adds or removes Pods based on the change to its `.spec`.

Since each pod requests 200 milli-cores by `kubectl run`, this means an average CPU usage of 100 milli-cores. See [Algorithm details](#) for more details on the algorithm.

Create the HorizontalPodAutoscaler:

```
kubectl autoscale deployment php-apache --cpu-percent=50 --min=1 --max=10
```

```
horizontalpodautoscaler.autoscaling/php-apache autoscaled
```

You can check the current status of the newly-made HorizontalPodAutoscaler, by running:

```
# You can use "hpa" or "horizontalpodautoscaler"; either name works OK
kubectl get hpa
```

The output is similar to:

NAME	REFERENCE	TARGET	MINPODS	MAXPODS
php-apache	Deployment/php-apache/scale	0% / 50%	1	10

(if you see other HorizontalPodAutoscalers with different names, that means they already existed, and isn't usually a problem).

Please note that the current CPU consumption is 0% as there are no clients sending requests to the server (the `TARGET` column shows the average across all the Pods controlled by the corresponding deployment).

Increase the load

Next, see how the autoscaler reacts to increased load. To do this, you'll start a different Pod to act as a client. The container within the client Pod runs in an infinite loop, sending queries to the php-apache service.

```
# Run this in a separate terminal
# so that the load generation continues and you can carry on with the next steps
kubectl run -i --tty load-generator --rm --image=busybox:1.28 --restart=Never
```

Now run:

```
# type Ctrl+C to end the watch when you're ready
kubectl get hpa php-apache --watch
```

Within a minute or so, you should see the higher CPU load; for example:

NAME	REFERENCE	TARGET	MINPODS	MAXPOD
php-apache	Deployment/php-apache/scale	305% / 50%	1	10

and then, more replicas. For example:

NAME	REFERENCE	TARGET	MINPODS	MAXPOD
php-apache	Deployment/php-apache/scale	305% / 50%	1	10

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
```

You should see the replica count matching the figure from the HorizontalPodAutoscaler

NAME	READY	UP-TO-DATE	AVAILABLE	AGE
php-apache	7/7	7	7	19m

Note: It may take a few minutes to stabilize the number of replicas. Since the amount of load is not controlled in any way it may happen that the final number of replicas will differ from this example.

Stop generating load

To finish the example, stop sending the load.

In the terminal where you created the Pod that runs a `busybox` image, terminate the load generation by typing `<ctrl> + C`.

Then verify the result state (after a minute or so):

```
# type Ctrl+C to end the watch when you're ready
kubectl get hpa php-apache --watch
```

The output is similar to:

NAME	REFERENCE	TARGET	MINPODS	MAXPOD
php-apache	Deployment/php-apache/scale	0% / 50%	1	10

and the Deployment also shows that it has scaled down:

```
kubectl get deployment php-apache
```

NAME	READY	UP-TO-DATE	AVAILABLE	AGE
php-apache	1/1	1	1	27m

Once CPU utilization dropped to 0, the HPA automatically scaled the number of replicas back down to 1.

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/v2` API version.

First, get the YAML of your HorizontalPodAutoscaler in the `autoscaling/v2` form:

```
kubectl get hpa php-apache -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/v2
kind: HorizontalPodAutoscaler
metadata:
  name: php-apache
spec:
  scaleTargetRef:
    apiVersion: apps/v1
    kind: Deployment
    name: php-apache
  minReplicas: 1
  maxReplicas: 10
  metrics:
  - type: Resource
    resource:
      name: cpu
    target:
      type: Utilization
      averageUtilization: 50
status:
  observedGeneration: 1
  lastScaleTime: <some-time>
  currentReplicas: 1
  desiredReplicas: 1
  currentMetrics:
  - type: Resource
    resource:
      name: cpu
    current:
      averageUtilization: 0
      averageValue: 0
```

Notice that the `targetCPUUtilizationPercentage` field has been replaced with an array called `metrics`. The CPU utilization metric is a *resource metric*, since it is represented as a percentage of a resource specified on pod containers. Notice that you can specify other resource metrics besides CPU. By default, the only other supported resource metric is memory. These resources do not change names from cluster to cluster, and should always be available, as long as the `metrics.k8s.io` API is available.

You can also specify resource metrics in terms of direct values, instead of as percentages of the requested value, by using a `target.type` of `AverageValue` instead of `utilization`, and setting the corresponding `target.averageValue` field instead of the `target.averageUtilization`.

There are two other types of metrics, both of which are considered *custom metrics*: pod metrics and object metrics. These metrics may have names which are cluster specific, and require a more advanced cluster monitoring setup.

The first of these alternative metric types is *pod metrics*. These metrics describe Pods, and are averaged together across Pods and compared with a target value to determine the replica count. They work much like resource metrics, except that they *only* support a `target` type of `AverageValue`.

Pod metrics are specified using a metric block like this:

```
type: Pods
pods:
  metric:
    name: packets-per-second
  target:
    type: AverageValue
    averageValue: 1k
```

The second alternative metric type is *object metrics*. These metrics describe a different object in the same namespace, instead of describing Pods. The metrics are not necessarily fetched from the object; they only describe it. Object metrics support `target` types of both `value` and `AverageValue`. With `value`, the target is compared directly to the returned metric from the API. With `AverageValue`, the value returned from the custom metrics API is divided by the number of Pods before being compared to the target. The following example is the YAML representation of the `requests-per-second` metric.

```
type: Object
object:
  metric:
    name: requests-per-second
  describedObject:
    apiVersion: networking.k8s.io/v1
    kind: Ingress
    name: main-route
  target:
    type: Value
    value: 2k
```

If you provide multiple such metric blocks, the HorizontalPodAutoscaler will consider each metric in turn. The HorizontalPodAutoscaler will calculate proposed replica counts for each metric, and then choose the one with the highest replica count.

For example, if you had your monitoring system collecting metrics about network traffic, you could update the definition above using `kubectl edit` to look like this:

```
apiVersion: autoscaling/v2
kind: HorizontalPodAutoscaler
metadata:
  name: php-apache
spec:
  scaleTargetRef:
    apiVersion: apps/v1
    kind: Deployment
    name: php-apache
  minReplicas: 1
  maxReplicas: 10
  metrics:
    - type: Resource
      resource:
        name: cpu
      target:
        type: Utilization
        averageUtilization: 50
    - type: Pods
      pods:
        metric:
          name: packets-per-second
        target:
          type: AverageValue
          averageValue: 1k
    - type: Object
      object:
        metric:
          name: requests-per-second
        describedObject:
          apiVersion: networking.k8s.io/v1
          kind: Ingress
          name: main-route
        target:
          type: Value
          value: 10k
  status:
    observedGeneration: 1
    lastScaleTime: <some-time>
    currentReplicas: 1
    desiredReplicas: 1
    currentMetrics:
      - type: Resource
        resource:
          name: cpu
        current:
          averageUtilization: 0
          averageValue: 0
      - type: Object
        object:
          metric:
            name: requests-per-second
        describedObject:
          apiVersion: networking.k8s.io/v1
          kind: Ingress
          name: main-route
        current:
          value: 10k
```

Then, your HorizontalPodAutoscaler would attempt to ensure that each pod was consuming roughly 50% of its requested CPU, serving 1000 packets per second, and that all pods behind the main-route Ingress were serving a total of 10000 requests per second.

Autoscaling on more specific metrics

Many metrics pipelines allow you to describe metrics either by name or by a set of additional descriptors called *labels*. For all non-resource metric types (pod, object, and external, described below), you can specify an additional label selector which is passed to your metric pipeline. For instance, if you collect a metric `http_requests` with the `verb` label, you can specify the following metric block to scale only on GET requests:

```
type: Object
object:
  metric:
    name: http_requests
    selector: {matchLabels: {verb: GET}}
```

This selector uses the same syntax as the full Kubernetes label selectors. The monitoring pipeline determines how to collapse multiple series into a single value, if the name and selector match multiple series. The selector is additive, and cannot select metrics that describe objects that are **not** the target object (the target pods in the case of the `Pods` type, and the described object in the case of the `Object` type).

Autoscaling on metrics not related to Kubernetes objects

Applications running on Kubernetes may need to autoscale based on metrics that don't have an obvious relationship to any object in the Kubernetes cluster, such as metrics describing a hosted service with no direct correlation to Kubernetes namespaces. In Kubernetes 1.10 and later, you can address this use case with *external metrics*.

Using external metrics requires knowledge of your monitoring system; the setup is similar to that required when using custom metrics. External metrics allow you to autoscale your cluster based on any metric available in your monitoring system. Provide a `metric` block with a `name` and `selector`, as above, and use the `External` metric type instead of `object`. If multiple time series are matched by the `metricSelector`, the sum of their values is used by the HorizontalPodAutoscaler. External metrics support both the `value` and `AverageValue` target types, which function exactly the same as when you use the `Object` type.

For example if your application processes tasks from a hosted queue service, you could add the following section to your HorizontalPodAutoscaler manifest to specify that you need one worker per 30 outstanding tasks.

```
- type: External
  external:
    metric:
      name: queue_messages_ready
      selector:
        matchLabels:
          queue: "worker_tasks"
    target:
      type: AverageValue
      averageValue: 30
```

When possible, it's preferable to use the custom metric target types instead of external metrics, since it's easier for cluster administrators to secure the custom metrics API. The external metrics API potentially allows

access to any metric, so cluster administrators should take care when exposing it.

Appendix: Horizontal Pod Autoscaler Status Conditions

When using the `autoscaling/v2` form of the HorizontalPodAutoscaler, you will be able to see *status conditions* set by Kubernetes on the HorizontalPodAutoscaler. These status conditions indicate whether or not the HorizontalPodAutoscaler is able to scale, and whether or not it is currently restricted in any way.

The conditions appear in the `status.conditions` field. To see the conditions affecting a HorizontalPodAutoscaler, we can use `kubectl describe hpa`:

```
kubectl describe hpa cm-test
```

Name:	cm-test		
Namespace:	prom		
Labels:	<none>		
Annotations:	<none>		
CreationTimestamp:	Fri, 16 Jun 2017 18:09:22 +0000		
Reference:	ReplicationController/cm-test		
Metrics:	(current / target)		
"http_requests" on pods:	66m / 500m		
Min replicas:	1		
Max replicas:	4		
ReplicationController pods:	1 current / 1 desired		
Conditions:			
Type	Status	Reason	Message
-----	-----	-----	-----
AbleToScale	True	ReadyForNewScale	the last scale
ScalingActive	True	ValidMetricFound	the HPA was abl
ScalingLimited	False	DesiredWithinRange	the desired rep
Events:			

For this HorizontalPodAutoscaler, you 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.

Quantities

All metrics in the HorizontalPodAutoscaler and metrics APIs are specified using a special whole-number notation known in Kubernetes as a quantity. For example, the quantity `10500m` would be written as `10.5` in decimal notation. The metrics APIs will return whole numbers without a suffix when possible, and will generally return quantities in milli-units otherwise. This means you might see your metric value fluctuate between `1` and `1500m`, or `1` and `1.5` when written in decimal notation.

Other possible scenarios

Creating the autoscaler declaratively

Instead of using `kubectl autoscale` command to create a HorizontalPodAutoscaler imperatively we can use the following manifest to create it declaratively:

```
application/hpa/php-apache.yaml ┌─
```

```
apiVersion: autoscaling/v1
kind: HorizontalPodAutoscaler
metadata:
  name: php-apache
spec:
  scaleTargetRef:
    apiVersion: apps/v1
    kind: Deployment
    name: php-apache
  minReplicas: 1
  maxReplicas: 10
  targetCPUUtilizationPercentage: 50
```

Then, create the autoscaler by executing the following command:

```
kubectl create -f https://k8s.io/examples/application/hpa/php-apache.y
◀ ────────────────────────────────────────────────────────────────────────── ▶
```

```
horizontalpodautoscaler.autoscaling/php-apache created
```

8.9 - Specifying a Disruption Budget for your Application

FEATURE STATE: [Kubernetes v1.21 \[stable\]](#)

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

Your Kubernetes server must be at or later than version v1.21. To check the version, enter `kubectl version`.

- You are the owner of an application running on a Kubernetes cluster that requires high availability.
- You should know how to deploy [Replicated Stateless Applications](#) and/or [Replicated Stateful Applications](#).
- You should have read about [Pod Disruptions](#).
- You should confirm with your cluster owner or service provider that they respect Pod Disruption Budgets.

Protecting an Application with a PodDisruptionBudget

1. Identify what application you want to protect with a PodDisruptionBudget (PDB).
2. Think about how your application reacts to disruptions.
3. Create a PDB definition as a YAML file.
4. Create the PDB object from the YAML file.

Identify an Application to Protect

The most common use case when you want to protect an application specified by one of the built-in Kubernetes controllers:

- Deployment
- ReplicationController
- ReplicaSet
- StatefulSet

In this case, make a note of the controller's `.spec.selector`; the same selector goes into the PDBs `.spec.selector`.

From version 1.15 PDBs support custom controllers where the [scale subresource](#) is enabled.

You can also use PDBs with pods which are not controlled by one of the above controllers, or arbitrary groups of pods, but there are some restrictions, described in [Arbitrary workloads and arbitrary selectors](#).

Think about how your application reacts to disruptions

Decide how many instances can be down at the same time for a short period due to a voluntary disruption.

- Stateless frontends:
 - Concern: don't reduce serving capacity by more than 10%.
 - Solution: use PDB with minAvailable 90% for example.
- Single-instance Stateful Application:
 - Concern: do not terminate this application without talking to me.
 - Possible Solution 1: Do not use a PDB and tolerate occasional downtime.
 - Possible Solution 2: Set PDB with maxUnavailable=0. Have an understanding (outside of Kubernetes) that the cluster operator needs to consult you before termination. When the cluster operator contacts you, prepare for downtime, and then delete the PDB to indicate readiness for disruption. Recreate afterwards.
- Multiple-instance Stateful application such as Consul, ZooKeeper, or etcd:
 - Concern: Do not reduce number of instances below quorum, otherwise writes fail.
 - Possible Solution 1: set maxUnavailable to 1 (works with varying scale of application).
 - Possible Solution 2: set minAvailable to quorum-size (e.g. 3 when scale is 5). (Allows more disruptions at once).
- Restartable Batch Job:
 - Concern: Job needs to complete in case of voluntary disruption.
 - Possible solution: Do not create a PDB. The Job controller will create a replacement pod.

Rounding logic when specifying percentages

Values for `minAvailable` or `maxUnavailable` can be expressed as integers or as a percentage.

- When you specify an integer, it represents a number of Pods. For instance, if you set `minAvailable` to 10, then 10 Pods must always be available, even during a disruption.
- When you specify a percentage by setting the value to a string representation of a percentage (eg. "50%"), it represents a percentage of total Pods. For instance, if you set `minAvailable` to "50%", then at least 50% of the Pods remain available during a disruption.

When you specify the value as a percentage, it may not map to an exact number of Pods. For example, if you have 7 Pods and you set `minAvailable` to "50%", it's not immediately obvious whether that means 3 Pods or 4 Pods must be available. Kubernetes rounds up to the nearest integer, so in this case, 4 Pods must be available. When you specify the value `maxUnavailable` as a percentage, Kubernetes rounds up the number of Pods that may be disrupted. Thereby a disruption can exceed your defined `maxUnavailable` percentage. You can examine the [code](#) that controls this behavior.

Specifying a PodDisruptionBudget

A `PodDisruptionBudget` has three fields:

- A label selector `.spec.selector` to specify the set of pods to which it applies. This field is required.
- `.spec.minAvailable` which is a description of the number of pods from that set that must still be available after the eviction, even in the absence of the evicted pod. `minAvailable` can be either an absolute number or a percentage.

- `.spec.maxUnavailable` (available in Kubernetes 1.7 and higher) which is a description of the number of pods from that set that can be unavailable after the eviction. It can be either an absolute number or a percentage.

Note: The behavior for an empty selector differs between the policy/v1beta1 and policy/v1 APIs for PodDisruptionBudgets. For policy/v1beta1 an empty selector matches zero pods, while for policy/v1 an empty selector matches every pod in the namespace.

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 the number of unhealthy replicas does not exceed 30% of the total number of desired replica rounded up to the nearest integer. If the total number of desired replicas is just one, that single replica is still allowed for disruption, leading to an effective unavailability of 100%.

In typical usage, a single budget would be used for a collection of pods managed by a controller—for example, the pods in a single ReplicaSet or StatefulSet.

Note: A disruption budget does not truly guarantee that the specified number/percentage of pods will always be up. For example, a node that hosts a pod from the collection may fail when the collection is at the minimum size specified in the budget, thus bringing the number of available pods from the collection below the specified size. The budget can only protect against voluntary evictions, not all causes of unavailability.

If you set `maxUnavailable` to 0% or 0, or you set `minAvailable` to 100% or the number of replicas, you are requiring zero voluntary evictions. When you set zero voluntary evictions for a workload object such as ReplicaSet, then you cannot successfully drain a Node running one of those Pods. If you try to drain a Node where an unevictable Pod is running, the drain never completes. This is permitted as per the semantics of `PodDisruptionBudget`.

You can find examples of pod disruption budgets defined below. They match pods with the label `app: zookeeper`.

Example PDB Using minAvailable:

[policy/zookeeper-pod-disruption-budget-minavailable.yaml](#) 

```
apiVersion: policy/v1
kind: PodDisruptionBudget
metadata:
  name: zk-pdb
spec:
  minAvailable: 2
  selector:
    matchLabels:
      app: zookeeper
```

Example PDB Using maxUnavailable:

[policy/zookeeper-pod-disruption-budget-maxunavailable.yaml](#) 

```
apiVersion: policy/v1
kind: PodDisruptionBudget
metadata:
  name: zk-pdb
spec:
  maxUnavailable: 1
  selector:
    matchLabels:
      app: zookeeper
```

For example, if the above `zk-pdb` object selects the pods of a StatefulSet of size 3, both specifications have the exact same meaning. The use of `maxUnavailable` is recommended as it automatically responds to changes in the number of replicas of the corresponding controller.

Create the PDB object

You can create or update the PDB object using kubectl.

```
kubectl apply -f mypdb.yaml
```

Check the status of the PDB

Use kubectl to check that your PDB is created.

Assuming you don't actually have pods matching `app: zookeeper` in your namespace, then you'll see something like this:

```
kubectl get poddisruptionbudgets
```

NAME	MIN AVAILABLE	MAX UNAVAILABLE	ALLOWED DISRUPTIONS	AGE
zk-pdb	2	N/A	0	7s

If there are matching pods (say, 3), then you would see something like this:

```
kubectl get poddisruptionbudgets
```

NAME	MIN AVAILABLE	MAX UNAVAILABLE	ALLOWED DISRUPTIONS	AGE
zk-pdb	2	N/A	1	7s

The non-zero value for `ALLOWED DISRUPTIONS` means that the disruption controller has seen the pods, counted the matching pods, and updated the status of the PDB.

You can get more information about the status of a PDB with this command:

```
kubectl get poddisruptionbudgets zk-pdb -o yaml
```

```
apiVersion: policy/v1
kind: PodDisruptionBudget
metadata:
  annotations:
...
  creationTimestamp: "2020-03-04T04:22:56Z"
  generation: 1
  name: zk-pdb
...
status:
  currentHealthy: 3
  desiredHealthy: 2
  disruptionsAllowed: 1
  expectedPods: 3
  observedGeneration: 1
```

Healthiness of a Pod

The current implementation considers healthy pods, as pods that have `.status.conditions` item with `type="Ready"` and `status="True"`. These pods are tracked via `.status.currentHealthy` field in the PDB status.

Unhealthy Pod Eviction Policy

FEATURE STATE: [Kubernetes v1.27 \[beta\]](#)

Note: This feature is enabled by default. You can disable it by disabling the [PDBUnhealthyPodEvictionPolicy feature gate](#) on the API server.

PodDisruptionBudget guarding an application ensures that `.status.currentHealthy` number of pods does not fall below the number specified in `.status.desiredHealthy` by disallowing eviction of healthy pods. By using `.spec.unhealthyPodEvictionPolicy`, you can also define the criteria when unhealthy pods should be considered for eviction. The default behavior when no policy is specified corresponds to the `IfHealthyBudget` policy.

Policies:

[IfHealthyBudget](#)

Running pods (`.status.phase="Running"`), but not yet healthy can be evicted only if the guarded application is not disrupted (`.status.currentHealthy` is at least equal to `.status.desiredHealthy`).

This policy ensures that running pods of an already disrupted application have the best chance to become healthy. This has negative implications for draining nodes, which can be blocked by misbehaving applications that are guarded by a PDB. More specifically applications with pods in `CrashLoopBackOff` state (due to a bug or misconfiguration), or pods that are just failing to report the `Ready` condition.

AlwaysAllow

Running pods (`.status.phase="Running"`), but not yet healthy are considered disrupted and can be evicted regardless of whether the criteria in a PDB is met.

This means prospective running pods of a disrupted application might not get a chance to become healthy. By using this policy, cluster managers can easily evict misbehaving applications that are guarded by a PDB. More specifically applications with pods in `CrashLoopBackOff` state (due to a bug or misconfiguration), or pods that are just failing to report the `Ready` condition.

Note: Pods in `Pending`, `Succeeded` or `Failed` phase are always considered for eviction.

Arbitrary workloads and arbitrary selectors

You can skip this section if you only use PDBs with the built-in workload resources (Deployment, ReplicaSet, StatefulSet and ReplicationController) or with custom resources that implement a `scale` [subresource](#), and where the PDB selector exactly matches the selector of the Pod's owning resource.

You can use a PDB with pods controlled by another resource, 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.

It is not possible to use other availability configurations, because Kubernetes cannot derive a total number of pods without a supported owning resource.

You can use a selector which selects a subset or superset of the pods belonging to a workload resource. The eviction API will disallow eviction of any pod covered by multiple PDBs, so most users will want to avoid overlapping selectors. One reasonable use of overlapping PDBs is when pods are being transitioned from one PDB to another.

8.10 - Accessing the Kubernetes API from a Pod

This guide demonstrates how to access the Kubernetes API from within a pod.

Before you begin

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

- [Killercoda](#)
- [Play with Kubernetes](#)

Accessing the API from within a Pod

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

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

Using Official Client Libraries

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

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

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

Directly accessing the REST API

While running in a Pod, your container can create an HTTPS URL for the Kubernetes API server by fetching the `KUBERNETES_SERVICE_HOST` and `KUBERNETES_SERVICE_PORT_HTTPS` environment variables. The API server's in-cluster address is also published to a Service named `kubernetes` in the `default` namespace so that pods may reference `kubernetes.default.svc` as a DNS name for the local API server.

Note: Kubernetes does not guarantee that the API server has a valid certificate for the hostname `kubernetes.default.svc`; however, the

control plane **is** expected to present a valid certificate for the hostname or IP address that `$KUBERNETES_SERVICE_HOST` represents.

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

```
/var/run/secrets/kubernetes.io/serviceaccount/token .
```

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

Finally, the default namespace to be used for namespaced API operations is placed in a file at

```
/var/run/secrets/kubernetes.io/serviceaccount/namespace
```

in each container.

Using kubectl proxy

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

Without using a proxy

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

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

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

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

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

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

# Explore the API with TOKEN
curl --cacert ${CACERT} --header "Authorization: Bearer ${TOKEN}" -X GET https://$APISERVER/api/v1/namespaces/$NAMESPACE/services/https:kubernetes.default.svc/proxy/
```

The output will be similar to this:

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

9 - Run Jobs

Run Jobs using parallel processing.

9.1 - Running Automated Tasks with a CronJob

This page shows how to run automated tasks using Kubernetes [CronJob](#) object.

Before you begin

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

Creating a CronJob

Cron jobs require a config file. Here is a manifest for a CronJob that runs a simple demonstration task every minute:

```
application/job/cronjob.yaml 
```

```
apiVersion: batch/v1
kind: CronJob
metadata:
  name: hello
spec:
  schedule: "* * * * *"
  jobTemplate:
    spec:
      template:
        spec:
          containers:
            - name: hello
              image: busybox:1.28
              imagePullPolicy: IfNotPresent
              command:
                - /bin/sh
                - -c
                - date; echo Hello from the Kubernetes cluster
  restartPolicy: OnFailure
```

Run the example CronJob by using this command:

```
kubectl create -f https://k8s.io/examples/application/job/cronjob.yaml
```

The output is similar to this:

```
cronjob.batch/hello created
```

After creating the cron job, get its status using this command:

```
kubectl get cronjob hello
```

The output is similar to this:

NAME	SCHEDULE	SUSPEND	ACTIVE	LAST SCHEDULE	AGE
hello	*/1 * * * *	False	0	<none>	10s

As you can see from the results of the command, the cron job has not scheduled or run any jobs yet. Watch for the job to be created in around one minute:

```
kubectl get jobs --watch
```

The output is similar to this:

NAME	COMPLETIONS	DURATION	AGE
hello-4111706356	0/1	0s	
hello-4111706356	0/1	0s	
hello-4111706356	1/1	5s	

Now you've seen one running job scheduled by the "hello" cron job. You can stop watching the job and view the cron job again to see that it scheduled the job:

```
kubectl get cronjob hello
```

The output is similar to this:

NAME	SCHEDULE	SUSPEND	ACTIVE	LAST SCHEDULE	AGE
hello	*/1 * * * *	False	0	50s	75s

You should see that the cron job `hello` successfully scheduled a job at the time specified in `LAST SCHEDULE`. There are currently 0 active jobs, meaning that the job has completed or failed.

Now, find the pods that the last scheduled job created and view the standard output of one of the pods.

Note: The job name is different from the pod name.

```
# Replace "hello-4111706356" with the job name in your system
pods=$(kubectl get pods --selector=job-name=hello-4111706356 --output=
```

Show the pod log:

```
kubectl logs $pods
```

The output is similar to this:

```
Fri Feb 22 11:02:09 UTC 2019  
Hello from the Kubernetes cluster
```

Deleting a CronJob

When you don't need a cron job any more, delete it with `kubectl delete cronjob <cronjob name>`:

```
kubectl delete cronjob hello
```

Deleting the cron job removes all the jobs and pods it created and stops it from creating additional jobs. You can read more about removing jobs in [garbage collection](#).

9.2 - 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 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 exits.

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. It is recommended to run this tutorial on a cluster with at least two nodes that are not acting as control plane hosts. If you do not already have a cluster, you can create one by using [minikube](#) or you can use one of these Kubernetes playgrounds:

- [Killercoda](#)
- [Play with Kubernetes](#)

Starting a message queue service

This example uses RabbitMQ, however, you can adapt the example to use another AMQP-type message service.

In practice you could set up a message queue service once in a cluster and reuse it for many jobs, as well as for long-running services.

Start RabbitMQ as follows:

```
kubectl create -f https://raw.githubusercontent.com/kubernetes/kuberne
```

```
service "rabbitmq-service" created
```

```
kubectl create -f https://raw.githubusercontent.com/kubernetes/kuberne
```

```
replicationcontroller "rabbitmq-controller" created
```

We will only use the rabbitmq part from the [celery-rabbitmq example](#).

Testing the message queue service

Now, we can experiment with accessing the message queue. We will create a temporary interactive pod, install some tools on it, and experiment with queues.

First create a temporary interactive Pod.

```
# Create a temporary interactive container
kubectl run -i --tty temp --image ubuntu:18.04
```

```
Waiting for pod default/temp-loe07 to be running, status is Pending, po
... [ 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 |
.... [ 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 set up 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
# 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_NAME:5672/

# Now create a queue:

root@temp-loe07:/# /usr/bin/amqp-declare-queue --url=$BROKER_URL -q foo
foo

# Publish one message to it:

root@temp-loe07:/# /usr/bin/amqp-publish --url=$BROKER_URL -r foo -p -c 1
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` prints out the characters read from standard input, and the echo adds a carriage return so the example is readable.

Filling the Queue with tasks

Now let's fill the queue with some "tasks". In our example, our tasks are strings to be printed.

In practice, the content of the messages might be:

- names of files to that need to be processed
- extra flags to the program
- ranges of keys in a database table
- configuration parameters to a simulation
- frame numbers of a scene to be rendered

In practice, if there is large data that is needed in a read-only mode by all pods of the Job, you will typically put that in a shared file system like NFS and mount that readonly on all the pods, or the program in the pod will natively read data from a cluster file system like HDFS.

For our example, we will create the queue and fill it using the amqp command line tools. In practice, you might write a program to fill the queue using an amqp client library.

```
/usr/bin/amqp-declare-queue --url=$BROKER_URL -q job1 -d
job1
```

```
for f in apple banana cherry date fig grape lemon melon
do
  /usr/bin/amqp-publish --url=$BROKER_URL -r job1 -p -b $f
done
```

So, we filled the queue with 8 messages.

Create an Image

Now we are ready to create an image that we will run as a job.

We will use the `amqp-consume` utility to read the message from the queue and run our actual program. Here is a very simple example program:

[application/job/rabbitmq/worker.py](#)

```
#!/usr/bin/env python

# Just prints standard out and sleeps for 10 seconds.
import sys
import time
print("Processing " + sys.stdin.readlines()[0])
time.sleep(10)
```

Give the script execution permission:

```
chmod +x worker.py
```

Now, build an image. If you are working in the source tree, then change directory to `examples/job/work-queue-1`. Otherwise, make a temporary directory, change to it, download the [Dockerfile](#), and [worker.py](#). In either case, build the image with this command:

```
docker build -t job-wq-1 .
```

For the [Docker Hub](#), tag your app image with your username and push to the Hub with the below commands. Replace `<username>` with your Hub username.

```
docker tag job-wq-1 <username>/job-wq-1
docker push <username>/job-wq-1
```

If you are using [Google Container Registry](#), tag your app image with your project ID, and push to GCR. Replace `<project>` with your project ID.

```
docker tag job-wq-1 gcr.io/<project>/job-wq-1
gcloud docker -- push gcr.io/<project>/job-wq-1
```

Defining a Job

Here is a job definition. You'll need to make a copy of the Job and edit the image to match the name you used, and call it `./job.yaml`.

[application/job/rabbitmq/job.yaml](#)

```
apiVersion: batch/v1
kind: Job
metadata:
  name: job-wq-1
spec:
  completions: 8
  parallelism: 2
  template:
    metadata:
      name: job-wq-1
    spec:
      containers:
        - name: c
          image: gcr.io/<project>/job-wq-1
          env:
            - name: BROKER_URL
              value: amqp://guest:guest@rabbitmq-service:5672
            - name: QUEUE
              value: job1
  restartPolicy: OnFailure
```

In this example, each pod works on one item from the queue and then exits. So, the completion count of the Job corresponds to the number of work items done. So we set, `.spec.completions: 8` for the example, since we put 8 items in the queue.

Running the Job

So, now run the Job:

```
kubectl apply -f ./job.yaml
```

You can wait for the Job to succeed, with a timeout:

```
# The check for condition name is case insensitive
kubectl wait --for=condition=complete --timeout=300s job/job-wq-1
```

Next, 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:	<none>					
Volumes:	<none>					
Events:						
FirstSeen	LastSeen	Count	From	SubobjectPath	Type	Reason
27s	27s	1	{job }		Normal	Succeeded
27s	27s	1	{job }		Normal	Succeeded
27s	27s	1	{job }		Normal	Succeeded
27s	27s	1	{job }		Normal	Succeeded
26s	26s	1	{job }		Normal	Succeeded
15s	15s	1	{job }		Normal	Succeeded
14s	14s	1	{job }		Normal	Succeeded
14s	14s	1	{job }		Normal	Succeeded

All the pods for that Job succeeded. Yay.

Alternatives

This approach has the advantage that you do not need to modify your "worker" program to be aware that there is a work queue.

It does require that you run a message queue service. If running a queue service is inconvenient, you may want to consider one of the other [job patterns](#).

This approach creates a pod for every work item. If your work items only take a few seconds, though, creating a Pod for every work item may add a lot of overhead. Consider another [example](#), that executes multiple work items per Pod.

In this example, we use the `amqp-consume` utility to read the message from the queue and run our actual program. This has the advantage that you do not need to modify your program to be aware of the queue. A [different example](#), shows how to communicate with the work queue using a client library.

Caveats

If the number of completions is set to less than the number of items in the queue, then not all items will be processed.

If the number of completions is set to more than the number of items in the queue, then the Job will not appear to be completed, even though all items in the queue have been processed. It will start additional pods which will block waiting for a message.

There is an unlikely race with this pattern. If the container is killed in between the time that the message is acknowledged by the amqp-consume command and the time that the container exits with success, or if the node crashes before the kubelet is able to post the success of the pod back to the api-server, then the Job will not appear to be complete, even though all items in the queue have been processed.

9.3 - 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 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

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

- [Killercoda](#)
- [Play with Kubernetes](#)

Be familiar with the basic, non-parallel, use of [Job](#).

Starting Redis

For this example, for simplicity, we will start a single instance of Redis. See the [Redis Example](#) for an example of deploying Redis scalably and redundantly.

You could also download the following files directly:

- [redis-pod.yaml](#)
- [redis-service.yaml](#)
- [Dockerfile](#)
- [job.yaml](#)
- [rediswq.py](#)
- [worker.py](#)

Filling the Queue with tasks

Now let's fill the queue with some "tasks". In our example, our tasks are 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,
Hit enter for command prompt
```

Now hit enter, start the redis CLI, and create a list with some work items in it.

```
# redis-cli -h redis
redis:6379> rpush job2 "apple"
(integer) 1
redis:6379> rpush job2 "banana"
(integer) 2
redis:6379> rpush job2 "cherry"
(integer) 3
redis:6379> rpush job2 "date"
(integer) 4
redis:6379> rpush job2 "fig"
(integer) 5
redis:6379> rpush job2 "grape"
(integer) 6
redis:6379> rpush job2 "lemon"
(integer) 7
redis:6379> rpush job2 "melon"
(integer) 8
redis:6379> rpush job2 "orange"
(integer) 9
redis:6379> lrange job2 0 -1
1) "apple"
2) "banana"
3) "cherry"
4) "date"
5) "fig"
6) "grape"
7) "lemon"
8) "melon"
9) "orange"
```

So, the list with key `job2` will be our work queue.

Note: if you do not have Kube DNS setup correctly, you may need to change the first step of the above block to `redis-cli -h $REDIS_SERVICE_HOST`.

Create an Image

Now we are ready to create an image that we will run.

We will use a python worker program with a redis client to read the messages from the message queue.

A simple Redis work queue client library is provided, called `rediswq.py` ([Download](#)).

The "worker" program in each Pod of the Job uses the work queue client library to get work. Here it is:

[application/job/redis/worker.py](#) 

```
#!/usr/bin/env python

import time
import rediswq

host="redis"
# Uncomment next two lines if you do not have Kube-DNS working.
# import os
# host = os.getenv("REDIS_SERVICE_HOST")

q = rediswq.RedisWQ(name="job2", host=host)
print("Worker with sessionID: " + q.sessionID())
print("Initial queue state: empty=" + str(q.empty()))
while not q.empty():
    item = q.lease(lease_secs=10, block=True, timeout=2)
    if item is not None:
        itemstr = item.decode("utf-8")
        print("Working on " + itemstr)
        time.sleep(10) # Put your actual work here instead of sleep.
        q.complete(item)
    else:
        print("Waiting for work")
print("Queue empty, exiting")
```

You could also download [worker.py](#), [rediswq.py](#), and [Dockerfile](#) files, then build the image:

```
docker build -t job-wq-2 .
```

Push the image

For the [Docker Hub](#), tag your app image with your username and push to the Hub with the below commands. Replace `<username>` with your Hub username.

```
docker tag job-wq-2 <username>/job-wq-2
docker push <username>/job-wq-2
```

You need to push to a public repository or [configure your cluster to be able to access your private repository](#).

If you are using [Google Container Registry](#), tag your app image with your project ID, and push to GCR. Replace `<project>` with your project ID.

```
docker tag job-wq-2 gcr.io/<project>/job-wq-2
gcloud docker -- push gcr.io/<project>/job-wq-2
```

Defining a Job

Here is the job definition:

[application/job/redis/job.yaml](#) 

```
apiVersion: batch/v1
kind: Job
metadata:
  name: job-wq-2
spec:
  parallelism: 2
  template:
    metadata:
      name: job-wq-2
    spec:
      containers:
        - name: c
          image: gcr.io/myproject/job-wq-2
      restartPolicy: OnFailure
```

Be sure to edit the job template to change `gcr.io/myproject` to your own path.

In this example, each pod works on several items from the queue and then exits when there are no more items. Since the workers themselves detect when the workqueue is empty, and the Job controller does not know about the workqueue, it relies on the workers to signal when they are done working. The workers signal that the queue is empty by exiting with success. So, as soon as any worker exits with success, the controller knows the work is done, and the Pods will exit soon. So, we set the completion count of the Job to 1. The job controller will wait for the other pods to complete too.

Running the Job

So, now run the Job:

```
kubectl apply -f ./job.yaml
```

Now wait a bit, then check on the job.

```
kubectl describe jobs/job-wq-2
Name:          job-wq-2
Namespace:     default
Selector:      controller-uid=b1c7e4e3-92e1-11e7-b85e-fa163ee3c11f
Labels:        controller-uid=b1c7e4e3-92e1-11e7-b85e-fa163ee3c11f
               job-name=job-wq-2
Annotations:   <none>
Parallelism:  2
Completions:  <unset>
Start Time:   Mon, 11 Jan 2016 17:07:59 -0800
Pods Statuses: 1 Running / 0 Succeeded / 0 Failed
Pod Template:
  Labels:      controller-uid=b1c7e4e3-92e1-11e7-b85e-fa163ee3c11f
               job-name=job-wq-2
  Containers:
    c:
      Image:       gcr.io/exampleproject/job-wq-2
      Port:        -
      Environment: <none>
      Mounts:      -
      Volumes:     -
  Events:
    FirstSeen  LastSeen  Count  From           SubobjectPath  Type
    -----  -----  -----  ----  -----
    33s       33s       1      {job-controller }  Normal        Normal
```

You can wait for the Job to succeed, with a timeout:

```
# The check for condition name is case insensitive
kubectl wait --for=condition=complete --timeout=300s job/job-wq-2
```

```
kubectl logs pods/job-wq-2-7r7b2
```

```
Worker with sessionID: bbd72d0a-9e5c-4dd6-abf6-416cc267991f
Initial queue state: empty=False
Working on banana
Working on date
Working on lemon
```

As you can see, one of our pods worked on several work units.

Alternatives

If running a queue service or modifying your containers to use a work queue is inconvenient, you may want to consider one of the other [job patterns](#).

If you have a continuous stream of background processing work to run, then consider running your background workers with a `ReplicaSet` instead, and consider running a background processing library such as <https://github.com/resque/resque>.

9.4 - Indexed Job for Parallel Processing with Static Work Assignment

FEATURE STATE: [Kubernetes v1.24 \[stable\]](#)

In this example, you will run a Kubernetes Job that uses multiple parallel worker processes. Each worker is a different container running in its own Pod. The Pods have an *index number* that the control plane sets automatically, which allows each Pod to identify which part of the overall task to work on.

The pod index is available in the annotation `batch.kubernetes.io/job-completion-index` as a string representing its decimal value. In order for the containerized task process to obtain this index, you can publish the value of the annotation using the [downward API](#) mechanism. For convenience, the control plane automatically sets the downward API to expose the index in the `JOB_COMPLETION_INDEX` environment variable.

Here is an overview of the steps in this example:

1. **Define a Job manifest using indexed completion.** The downward API allows you to pass the pod index annotation as an environment variable or file to the container.
2. **Start an [Indexed](#) Job based on that manifest.**

Before you begin

You should already 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. It is recommended to run this tutorial on a cluster with at least two nodes that are not acting as control plane hosts. If you do not already have a cluster, you can create one by using [minikube](#) or you can use one of these Kubernetes playgrounds:

- [Killercoda](#)
- [Play with Kubernetes](#)

Your Kubernetes server must be at or later than version v1.21. To check the version, enter `kubectl version`.

Choose an approach

To access the work item from the worker program, you have a few options:

1. Read the `JOB_COMPLETION_INDEX` environment variable. The Job controller automatically links this variable to the annotation containing the completion index.
2. Read a file that contains the completion index.
3. Assuming that you can't modify the program, you can wrap it with a script that reads the index using any of the methods above and converts it into something that the program can use as input.

For this example, imagine that you chose option 3 and you want to run the [rev](#) utility. This program accepts a file as an argument and prints its content reversed.

```
rev data.txt
```

You'll use the `rev` tool from the [busybox](#) container image.

As this is only an example, each Pod only does a tiny piece of work (reversing a short string). In a real workload you might, for example, create a Job that represents the task of producing 60 seconds of video based on scene data. Each work item in the video rendering Job would be to render a particular frame of that video clip. Indexed completion would mean that each Pod in the Job knows which frame to render and publish, by counting frames from the start of the clip.

Define an Indexed Job

Here is a sample Job manifest that uses `Indexed` completion mode:

```
application/job/indexed-job.yaml □
```

```
apiVersion: batch/v1
kind: Job
metadata:
  name: 'indexed-job'
spec:
  completions: 5
  parallelism: 3
  completionMode: Indexed
  template:
    spec:
      restartPolicy: Never
      initContainers:
        - name: 'input'
          image: 'docker.io/library/bash'
          command:
            - "bash"
            - "-c"
            - |
              items=(foo bar baz qux xyz)
              echo ${items[$JOB_COMPLETION_INDEX]} > /input/data.txt
      volumeMounts:
        - mountPath: /input
          name: input
      containers:
        - name: 'worker'
          image: 'docker.io/library/busybox'
          command:
            - "rev"
            - "/input/data.txt"
      volumeMounts:
        - mountPath: /input
          name: input
      volumes:
        - name: input
          emptyDir: {}
```

In the example above, you use the builtin `JOB_COMPLETION_INDEX` environment variable set by the Job controller for all containers. An [init container](#) maps the index to a static value and writes it to a file that is

shared with the container running the worker through an [emptyDir volume](#). Optionally, you can [define your own environment variable through the downward API](#) to publish the index to containers. You can also choose to load a list of values from a [ConfigMap as an environment variable or file](#).

Alternatively, you can directly [use the downward API to pass the annotation value as a volume file](#), like shown in the following example:

```
application/job/indexed-job-vol.yaml □

apiVersion: batch/v1
kind: Job
metadata:
  name: 'indexed-job'
spec:
  completions: 5
  parallelism: 3
  completionMode: Indexed
  template:
    spec:
      restartPolicy: Never
      containers:
        - name: 'worker'
          image: 'docker.io/library/busybox'
          command:
            - "rev"
            - "/input/data.txt"
          volumeMounts:
            - mountPath: /input
              name: input
          volumes:
            - name: input
          downwardAPI:
            items:
              - path: "data.txt"
                fieldRef:
                  fieldPath: metadata.annotations['batch.kubernetes.io/job']

      ◀ ▶
```

Running the Job

Now run the Job:

```
# This uses the first approach (relying on $JOB_COMPLETION_INDEX)
kubectl apply -f https://kubernetes.io/examples/application/job/indexed-job-vol.yaml
```

When you create this Job, the control plane creates a series of Pods, one for each index you specified. The value of `.spec.parallelism` determines how many can run at once whereas `.spec.completions` determines how many Pods the Job creates in total.

Because `.spec.parallelism` is less than `.spec.completions`, the control plane waits for some of the first Pods to complete before starting more of them.

You can wait for the Job to succeed, with a timeout:

```
# The check for condition name is case insensitive
kubectl wait --for=condition=complete --timeout=300s job/indexed-job
```

Now, describe the Job and check that it was successful.

```
kubectl describe jobs/indexed-job
```

The output is similar to:

```
Name: indexed-job
Namespace: default
Selector: controller-uid=bf865e04-0b67-483b-9a90-74cf4c3e756
Labels: controller-uid=bf865e04-0b67-483b-9a90-74cf4c3e756
        job-name=indexed-job
Annotations: <none>
Parallelism: 3
Completions: 5
Start Time: Thu, 11 Mar 2021 15:47:34 +0000
Pods Statuses: 2 Running / 3 Succeeded / 0 Failed
Completed Indexes: 0-2
Pod Template:
  Labels: controller-uid=bf865e04-0b67-483b-9a90-74cf4c3e756
           job-name=indexed-job
  Init Containers:
    input:
      Image: docker.io/library/bash
      Port: <none>
      Host Port: <none>
      Command:
        bash
        -c
        items=(foo bar baz qux xyz)
        echo ${items[$JOB_COMPLETION_INDEX]} > /input/data.txt

    Environment: <none>
    Mounts:
      /input from input (rw)
  Containers:
    worker:
      Image: docker.io/library/busybox
      Port: <none>
      Host Port: <none>
      Command:
        rev
        /input/data.txt
      Environment: <none>
      Mounts:
        /input from input (rw)
  Volumes:
    input:
      Type: EmptyDir (a temporary directory that shares a pod's lifetime)
      Medium:
      SizeLimit: <unset>
Events:
  Type  Reason          Age   From            Message
  ----  -----          ---   ----            -----
  Normal SuccessfulCreate 4s   job-controller  Created pod: indexed-
  Normal SuccessfulCreate 4s   job-controller  Created pod: indexed-
  Normal SuccessfulCreate 4s   job-controller  Created pod: indexed-
  Normal SuccessfulCreate 1s   job-controller  Created pod: indexed-
  Normal SuccessfulCreate 1s   job-controller  Created pod: indexed-
```

In this example, you run the Job with custom values for each index. You can inspect the output of one of the pods:

```
kubectl logs indexed-job-fdhq5 # Change this to match the name of a Pod
```

The output is similar to:

```
xuq
```

9.5 - Job with Pod-to-Pod Communication

In this example, you will run a Job in [Indexed completion mode](#) configured such that the pods created by the Job can communicate with each other using pod hostnames rather than pod IP addresses.

Pods within a Job might need to communicate among themselves. The user workload running in each pod could query the Kubernetes API server to learn the IPs of the other Pods, but it's much simpler to rely on Kubernetes' built-in DNS resolution.

Jobs in Indexed completion mode automatically set the pods' hostname to be in the format of `${jobName}-${completionIndex}`. You can use this format to deterministically build pod hostnames and enable pod communication *without* needing to create a client connection to the Kubernetes control plane to obtain pod hostnames/IPs via API requests.

This configuration is useful for use cases where pod networking is required but you don't want to depend on a network connection with the Kubernetes API server.

Before you begin

You should already be familiar with the basic use of [Job](#).

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

- [KillerCoda](#)
- [Play with Kubernetes](#)

Your Kubernetes server must be at or later than version v1.21. To check the version, enter `kubectl version`.

Note: If you are using MiniKube or a similar tool, you may need to take [extra steps](#) to ensure you have DNS.

Starting a Job with Pod-to-Pod Communication

To enable pod-to-pod communication using pod hostnames in a Job, you must do the following:

1. Set up a [headless Service](#) with a valid label selector for the pods created by your Job. The headless service must be in the same namespace as the Job. One easy way to do this is to use the `job-name: <your-job-name>` selector, since the `job-name` label will be automatically added by Kubernetes. This configuration will trigger the DNS system to create records of the hostnames of the pods running your Job.
2. Configure the headless service as subdomain service for the Job pods by including the following value in your Job template spec:

```
subdomain: <headless-svc-name>
```

Example

Below is a working example of a Job with pod-to-pod communication via pod hostnames enabled. The Job is completed only after all pods successfully ping each other using hostnames.

Note: In the Bash script executed on each pod in the example below, the pod hostnames can be prefixed by the namespace as well if the pod needs to be reached from outside the namespace.

```
apiVersion: v1
kind: Service
metadata:
  name: headless-svc
spec:
  clusterIP: None # clusterIP must be None to create a headless service
  selector:
    job-name: example-job # must match Job name
  ---
apiVersion: batch/v1
kind: Job
metadata:
  name: example-job
spec:
  completions: 3
  parallelism: 3
  completionMode: Indexed
  template:
    spec:
      subdomain: headless-svc # has to match Service name
      restartPolicy: Never
      containers:
        - name: example-workload
          image: bash:latest
          command:
            - bash
            - -c
            - |
              for i in 0 1 2
              do
                gotStatus="-1"
                wantStatus="0"
                while [ $gotStatus -ne $wantStatus ]
                do
                  ping -c 1 example-job-${i}.headless-svc > /dev/null 2>&1
                  gotStatus=$?
                if [ $gotStatus -ne $wantStatus ]; then
                  echo "Failed to ping pod example-job-${i}.headless-svc"
                  sleep 1
                fi
              done
              echo "Successfully pinged pod: example-job-${i}.headless-svc"
            done
```

After applying the example above, reach each other over the network using: <pod-hostname>.<headless-service-name>. You should see output similar to the following:

```
kubectl logs example-job-0-qws42
```

```
Failed to ping pod example-job-0.headless-svc, retrying in 1 second...
Successfully pinged pod: example-job-0.headless-svc
Successfully pinged pod: example-job-1.headless-svc
Successfully pinged pod: example-job-2.headless-svc
```

Note: Keep in mind that the <pod-hostname>.<headless-service-name> name format used in this example would not work with DNS policy set to `None` or `Default`. You can learn more about pod DNS policies [here](#).

9.6 - Parallel Processing using Expansions

This task demonstrates running multiple [Jobs](#) based on a common template. You can use this approach to process batches of work in parallel.

For this example there are only three items: *apple*, *banana*, and *cherry*. The sample Jobs process each item by printing a string then pausing.

See [using Jobs in real workloads](#) to learn about how this pattern fits more realistic use cases.

Before you begin

You should 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. It is recommended to run this tutorial on a cluster with at least two nodes that are not acting as control plane hosts. If you do not already have a cluster, you can create one by using [minikube](#) or you can use one of these Kubernetes playgrounds:

- [KillerCoda](#)
- [Play with Kubernetes](#)

For basic templating you need the command-line utility `sed`.

To follow the advanced templating example, you need a working installation of [Python](#), and the Jinja2 template library for Python.

Once you have Python set up, you can install Jinja2 by running:

```
pip install --user jinja2
```

Create Jobs based on a template

First, download the following template of a Job to a file called `job-tmpl.yaml`. Here's what you'll download:

[application/job/job tmpl.yaml](#) 

```

apiVersion: batch/v1
kind: Job
metadata:
  name: process-item-$ITEM
  labels:
    jobgroup: jobexample
spec:
  template:
    metadata:
      name: jobexample
      labels:
        jobgroup: jobexample
    spec:
      containers:
      - name: c
        image: busybox:1.28
        command: ["sh", "-c", "echo Processing item $ITEM && sleep 5"]
      restartPolicy: Never

```

```

# Use curl to download job-tmpl.yaml
curl -L -s -o https://k8s.io/examples/application/job/job-tmpl.yaml

```

The file you downloaded is not yet a valid Kubernetes [manifest](#). Instead that template is a YAML representation of a Job object with some placeholders that need to be filled in before it can be used. The `$ITEM` syntax is not meaningful to Kubernetes.

Create manifests from the template

The following shell snippet uses `sed` to replace the string `$ITEM` with the loop variable, writing into a temporary directory named `jobs`. Run this now:

```

# Expand the template into multiple files, one for each item to be processed
mkdir ./jobs
for i in apple banana cherry
do
  cat job-tmpl.yaml | sed "s/\$ITEM/$i/" > ./jobs/job-$i.yaml
done

```

Check if it worked:

```
ls jobs/
```

The output is similar to this:

```
job-apple.yaml
job-banana.yaml
job-cherry.yaml
```

You could use any type of template language (for example: Jinja2; ERB), or write a program to generate the Job manifests.

Create Jobs from the manifests

Next, create all the Jobs with one kubectl command:

```
kubectl create -f ./jobs
```

The output is similar to this:

```
job.batch/process-item-apple created
job.batch/process-item-banana created
job.batch/process-item-cherry created
```

Now, check on the jobs:

```
kubectl get jobs -l jobgroup=jobexample
```

The output is similar to this:

NAME	COMPLETIONS	DURATION	AGE
process-item-apple	1/1	14s	22s
process-item-banana	1/1	12s	21s
process-item-cherry	1/1	12s	20s

Using the `-l` option to kubectl selects only the Jobs that are part of this group of jobs (there might be other unrelated jobs in the system).

You can check on the Pods as well using the same label selector:

```
kubectl get pods -l jobgroup=jobexample
```

The output is similar to:

NAME	READY	STATUS	RESTARTS	AGE
process-item-apple-kixwv	0/1	Completed	0	4m
process-item-banana-wrsf7	0/1	Completed	0	4m
process-item-cherry-dnfu9	0/1	Completed	0	4m

We can use this single command to check on the output of all jobs at once:

```
kubectl logs -f -l jobgroup=jobexample
```

The output should be:

```
Processing item apple
Processing item banana
Processing item cherry
```

Clean up

```
# Remove the Jobs you created
# Your cluster automatically cleans up their Pods
kubectl delete job -l jobgroup=jobexample
```

Use advanced template parameters

In the [first example](#), each instance of the template had one parameter, and that parameter was also used in the Job's name. However, [names](#) are restricted to contain only certain characters.

This slightly more complex example uses the [Jinja template language](#) to generate manifests and then objects from those manifests, with a multiple parameters for each Job.

For this part of the task, you are going to use a one-line Python script to convert the template to a set of manifests.

First, copy and paste the following template of a Job object, into a file called `job.yaml.jinja2`:

```
{% set params = [ { "name": "apple", "url": "http://dbpedia.org/resource/Apple_(fruit)" },
                  { "name": "banana", "url": "http://dbpedia.org/resource/Banana" },
                  { "name": "cherry", "url": "http://dbpedia.org/resource/Cherry" } ] %}
{% 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:1.28
          command: ["sh", "-c", "echo Processing URL {{ url }} && sleep 5"]
      restartPolicy: Never
{% endfor %}
```

The above template defines two parameters for each Job object using a list of python dicts (lines 1-4). A `for` loop emits one Job manifest for each set of parameters (remaining lines).

This example relies on a feature of YAML. One YAML file can contain multiple documents (Kubernetes manifests, in this case), separated by `---` on a line by itself. You can pipe the output directly to `kubectl` to create the Jobs.

Next, use this one-line Python program to expand the template:

```
alias render_template='python -c "from jinja2 import Template; import
```

Use `render_template` to convert the parameters and template into a single YAML file containing Kubernetes manifests:

```
# This requires the alias you defined earlier
cat job.yaml.jinja2 | render_template > jobs.yaml
```

You can view `jobs.yaml` to verify that the `render_template` script worked correctly.

Once you are happy that `render_template` is working how you intend, you can pipe its output into `kubectl`:

```
cat job.yaml.jinja2 | render_template | kubectl apply -f -
```

Kubernetes accepts and runs the Jobs you created.

Clean up

```
# Remove the Jobs you created
# Your cluster automatically cleans up their Pods
kubectl delete job -l jobgroup=jobexample
```

Using Jobs in real workloads

In a real use case, each Job performs some substantial computation, such as rendering a frame of a movie, or processing a range of rows in a database. If you were rendering a movie you would set `$ITEM` to the frame number. If you were processing rows from a database table, you would set `$ITEM` to represent the range of database rows to process.

In the task, you ran a command to collect the output from Pods by fetching their logs. In a real use case, each Pod for a Job writes its output to durable storage before completing. You can use a PersistentVolume for each Job, or an external storage service. For example, if you are rendering frames for a movie, use HTTP to `PUT` the rendered frame data to a URL, using a different URL for each frame.

Labels on Jobs and Pods

After you create a Job, Kubernetes automatically adds additional labels that distinguish one Job's pods from another Job's pods.

In this example, each Job and its Pod template have a label:

```
jobgroup=jobexample .
```

Kubernetes itself pays no attention to labels named `jobgroup`. Setting a label for all the Jobs you create from a template makes it convenient to operate on all those Jobs at once. In the [first example](#) you used a template to create several Jobs. The template ensures that each Pod also gets the same label, so you can check on all Pods for these templated Jobs with a single command.

Note: The label key `jobgroup` is not special or reserved. You can pick your own labelling scheme. There are [recommended labels](#) that you can use if you wish.

Alternatives

If you plan to create a large number of Job objects, you may find that:

- Even using labels, managing so many Jobs is cumbersome.
- If you create many Jobs in a batch, you might place high load on the Kubernetes control plane. Alternatively, the Kubernetes API server could rate limit you, temporarily rejecting your requests with a 429 status.
- You are limited by a [resource quota](#) on Jobs: the API server permanently rejects some of your requests when you create a great deal of work in one batch.

There are other [job patterns](#) that you can use to process large amounts of work without creating very many Job objects.

You could also consider writing your own [controller](#) to manage Job objects automatically.

9.7 - Handling retriable and non-retriable pod failures with Pod failure policy

FEATURE STATE: [Kubernetes v1.26 \[beta\]](#)

This document shows you how to use the [Pod failure policy](#), in combination with the default [Pod backoff failure policy](#), to improve the control over the handling of container- or Pod-level failure within a [Job](#).

The definition of Pod failure policy may help you to:

- better utilize the computational resources by avoiding unnecessary Pod retries.
- avoid Job failures due to Pod disruptions (such [preemption](#), [API-initiated eviction](#) or [taint-based eviction](#)).

Before you begin

You should already be familiar with the basic use of [Job](#).

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

- [Killercoda](#)
- [Play with Kubernetes](#)

Your Kubernetes server must be at or later than version v1.25. To check the version, enter [kubectl version](#).

Ensure that the [feature gates](#) `PodDisruptionConditions` and `JobPodFailurePolicy` are both enabled in your cluster.

Using Pod failure policy to avoid unnecessary Pod retries

With the following example, you can learn how to use Pod failure policy to avoid unnecessary Pod restarts when a Pod failure indicates a non-retriable software bug.

First, create a Job based on the config:

[/controllers/job-pod-failure-policy-failjob.yaml](#) 

```
apiVersion: batch/v1
kind: Job
metadata:
  name: job-pod-failure-policy-failjob
spec:
  completions: 8
  parallelism: 2
  template:
    spec:
      restartPolicy: Never
      containers:
        - name: main
          image: docker.io/library/bash:5
          command: ["bash"]
          args:
            - -c
            - echo "Hello world! I'm going to exit with 42 to simulate a failure"
      backoffLimit: 6
      podFailurePolicy:
        rules:
          - action: FailJob
            onExitCodes:
              containerName: main
              operator: In
              values: [42]
```

by running:

```
kubectl create -f job-pod-failure-policy-failjob.yaml
```

After around 30s the entire Job should be terminated. Inspect the status of the Job by running:

```
kubectl get jobs -l job-name=job-pod-failure-policy-failjob -o yaml
```

In the Job status, see a job Failed condition with the field reason equal PodFailurePolicy . Additionally, the message field contains a more detailed information about the Job termination, such as: Container main for pod default/job-pod-failure-policy-failjob-8ckj8 failed with exit code 42 matching FailJob rule at index 0 .

For comparison, if the Pod failure policy was disabled it would take 6 retries of the Pod, taking at least 2 minutes.

Clean up

Delete the Job you created:

```
kubectl delete jobs/job-pod-failure-policy-failjob
```

The cluster automatically cleans up the Pods.

Using Pod failure policy to ignore Pod disruptions

With the following example, you can learn how to use Pod failure policy to ignore Pod disruptions from incrementing the Pod retry counter towards the `.spec.backoffLimit` limit.

Caution: Timing is important for this example, so you may want to read the steps before execution. In order to trigger a Pod disruption it is important to drain the node while the Pod is running on it (within 90s since the Pod is scheduled).

1. Create a Job based on the config:

```
/controllers/job-pod-failure-policy-ignore.yaml 
```

```
apiVersion: batch/v1
kind: Job
metadata:
  name: job-pod-failure-policy-ignore
spec:
  completions: 4
  parallelism: 2
  template:
    spec:
      restartPolicy: Never
      containers:
        - name: main
          image: docker.io/library/bash:5
          command: ["bash"]
          args:
            - -c
            - echo "Hello world! I'm going to exit with 0 (success)."
      backoffLimit: 0
      podFailurePolicy:
        rules:
        - action: Ignore
          onPodConditions:
          - type: DisruptionTarget
```

by running:

```
kubectl create -f job-pod-failure-policy-ignore.yaml
```

2. Run this command to check the `nodeName` the Pod is scheduled to:

```
nodeName=$(kubectl get pods -l job-name=job-pod-failure-policy-ig
```

3. Drain the node to evict the Pod before it completes (within 90s):

```
kubectl drain nodes/$nodeName --ignore-daemonsets --grace-period=
```

4. Inspect the `.status.failed` to check the counter for the Job is not incremented:

```
kubectl get jobs -l job-name=job-pod-failure-policy-ignore -o yaml
```

5. Uncordon the node:

```
kubectl uncordon nodes/$nodeName
```

The Job resumes and succeeds.

For comparison, if the Pod failure policy was disabled the Pod disruption would result in terminating the entire Job (as the `.spec.backoffLimit` is set to 0).

Cleaning up

Delete the Job you created:

```
kubectl delete jobs/job-pod-failure-policy-ignore
```

The cluster automatically cleans up the Pods.

Using Pod failure policy to avoid unnecessary Pod retries based on custom Pod Conditions

With the following example, you can learn how to use Pod failure policy to avoid unnecessary Pod restarts based on custom Pod Conditions.

Note: The example below works since version 1.27 as it relies on transitioning of deleted pods, in the `Pending` phase, to a terminal phase (see: [Pod Phase](#)).

1. First, create a Job based on the config:

```
/controllers/job-pod-failure-policy-config-issue.yaml
```

```
apiVersion: batch/v1
kind: Job
metadata:
  name: job-pod-failure-policy-config-issue
spec:
  completions: 8
  parallelism: 2
  template:
    spec:
      restartPolicy: Never
      containers:
        - name: main
          image: "non-existing-repo/non-existing-image:example"
  backoffLimit: 6
  podFailurePolicy:
    rules:
      - action: FailJob
        onPodConditions:
          - type: ConfigIssue
```

by running:

```
kubectl create -f job-pod-failure-policy-config-issue.yaml
```

Note that, the image is misconfigured, as it does not exist.

2. Inspect the status of the job's Pods by running:

```
kubectl get pods -l job-name=job-pod-failure-policy-config-issue
```

You will see output similar to this:

```
containerStatuses:
- image: non-existing-repo/non-existing-image:example
  ...
  state:
    waiting:
      message: Back-off pulling image "non-existing-repo/non-exis
      reason: ImagePullBackOff
    ...
  phase: Pending
```

Note that the pod remains in the `Pending` phase as it fails to pull the misconfigured image. This, in principle, could be a transient issue and the image could get pulled. However, in this case, the image does not exist so we indicate this fact by a custom condition.

3. Add the custom condition. First prepare the patch by running:

```
cat <<EOF > patch.yaml
status:
  conditions:
  - type: ConfigIssue
    status: "True"
    reason: "NonExistingImage"
    lastTransitionTime: "$(date -u +"%Y-%m-%dT%H:%M:%SZ")"
EOF
```

Second, select one of the pods created by the job by running:

```
podName=$(kubectl get pods -l job-name=job-pod-failure-policy-con-
```

Then, apply the patch on one of the pods by running the following command:

```
kubectl patch pod $podName --subresource=status --patch-file=patc
```

If applied successfully, you will get a notification like this:

```
pod/job-pod-failure-policy-config-issue-k6pvp patched
```

4. Delete the pod to transition it to `Failed` phase, by running the command:

```
kubectl delete pods/$podName
```

5. Inspect the status of the Job by running:

```
kubectl get jobs -l job-name=job-pod-failure-policy-config-issue
```

In the Job status, see a job `Failed` condition with the field `reason` equal `PodFailurePolicy`. Additionally, the `message` field contains a more detailed information about the Job termination, such as: `Pod default/job-pod-failure-policy-config-issue-k6pvp has condition ConfigIssue matching FailJob rule at index 0.`

Note: In a production environment, the steps 3 and 4 should be automated by a user-provided controller.

Cleaning up

Delete the Job you created:

```
kubectl delete jobs/job-pod-failure-policy-config-issue
```

The cluster automatically cleans up the Pods.

Alternatives

You could rely solely on the [Pod backoff failure policy](#), by specifying the Job's `.spec.backoffLimit` field. However, in many situations it is problematic to find a balance between setting a low value for `.spec.backoffLimit` to avoid unnecessary Pod retries, yet high enough to make sure the Job would not be terminated by Pod disruptions.

10 - Access Applications in a Cluster

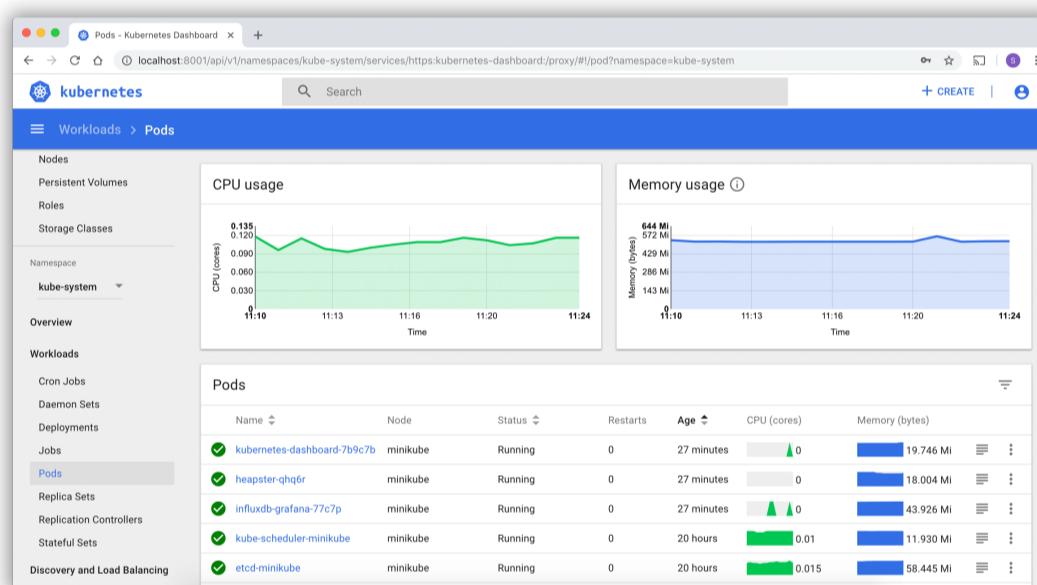
Configure load balancing, port forwarding, or setup firewall or DNS configurations to access applications in a cluster.

10.1 - Deploy and Access the Kubernetes Dashboard

Deploy the web UI (Kubernetes Dashboard) and access it.

Dashboard is a web-based Kubernetes user interface. You can use Dashboard to deploy containerized applications to a Kubernetes cluster, troubleshoot your containerized application, and manage the cluster resources. You can use Dashboard to get an overview of applications running on your cluster, as well as for creating or modifying individual Kubernetes resources (such as Deployments, Jobs, DaemonSets, etc). For example, you can scale a Deployment, initiate a rolling update, restart a pod or deploy new applications using a deploy wizard.

Dashboard also provides information on the state of Kubernetes resources in your cluster and on any errors that may have occurred.



Deploying the Dashboard UI

The Dashboard UI is not deployed by default. To deploy it, run the following command:

```
kubectl apply -f https://raw.githubusercontent.com/kubernetes/dashboard
```

Accessing the Dashboard UI

To protect your cluster data, Dashboard deploys with a minimal RBAC configuration by default. Currently, Dashboard only supports logging in with a Bearer Token. To create a token for this demo, you can follow our guide on [creating a sample user](#).

Warning: The sample user created in the tutorial will have administrative privileges and is for educational purposes only.

Command line proxy

You can enable access to the Dashboard using the `kubectl` command-line tool, by running the following command:

```
kubectl proxy
```

Kubectl will make Dashboard available at

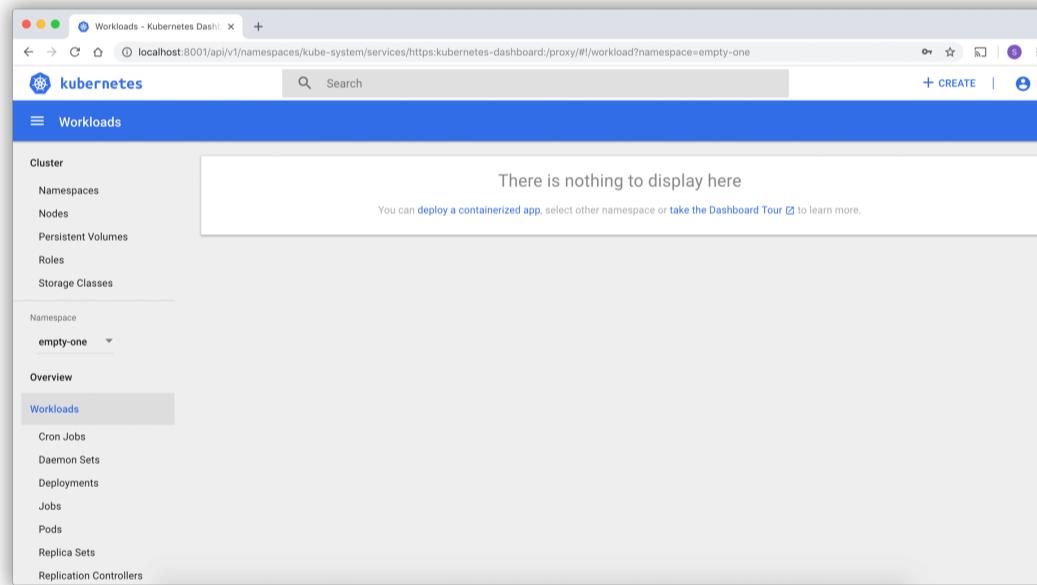
<http://localhost:8001/api/v1/namespaces/kubernetes-dashboard/services/https:kubernetes-dashboard:/proxy/>.

The UI can *only* be accessed from the machine where the command is executed. See `kubectl proxy --help` for more options.

Note: The kubeconfig authentication method does **not** support external identity providers or X.509 certificate-based authentication.

Welcome view

When you access Dashboard on an empty cluster, you'll see the welcome page. This page contains a link to this document as well as a button to deploy your first application. In addition, you can view which system applications are running by default in the `kube-system` [namespace](#) of your cluster, for example the Dashboard itself.



Deploying containerized applications

Dashboard lets you create and deploy a containerized application as a Deployment and optional Service with a simple wizard. You can either manually specify application details, or upload a YAML or JSON *manifest* file containing application configuration.

Click the **CREATE** button in the upper right corner of any page to begin.

Specifying application details

The deploy wizard expects that you provide the following information:

- **App name** (mandatory): Name for your application. A [label](#) with the name will be added to the Deployment and Service, if any, that will be deployed.

The application name must be unique within the selected Kubernetes [namespace](#). It must start with a lowercase character, and end with a lowercase character or a number, and contain only lowercase letters, numbers and dashes (-). It is limited to 24 characters. Leading and trailing spaces are ignored.

- **Container image** (mandatory): The URL of a public Docker [container image](#) on any registry, or a private image (commonly hosted on the Google Container Registry or Docker Hub). The container image specification must end with a colon.
- **Number of pods** (mandatory): The target number of Pods you want your application to be deployed in. The value must be a positive integer.

A [Deployment](#) will be created to maintain the desired number of Pods across your cluster.

- **Service** (optional): For some parts of your application (e.g. frontends) you may want to expose a [Service](#) onto an external, maybe public IP address outside of your cluster (external Service).

Note: For external Services, you may need to open up one or more ports to do so.

Other Services that are only visible from inside the cluster are called internal Services.

Irrespective of the Service type, if you choose to create a Service and your container listens on a port (incoming), you need to specify two ports. The Service will be created mapping the port (incoming) to the target port seen by the container. This Service will route to your deployed Pods. Supported protocols are TCP and UDP. The internal DNS name for this Service will be the value you specified as application name above.

If needed, you can expand the **Advanced options** section where you can specify more settings:

- **Description:** The text you enter here will be added as an [annotation](#) to the Deployment and displayed in the application's details.
- **Labels:** Default [labels](#) to be used for your application are application name and version. You can specify additional labels to be applied to the Deployment, Service (if any), and Pods, such as release, environment, tier, partition, and release track.

Example:

```
release=1.0
tier=frontend
environment=prod
track=stable
```

- **Namespace:** Kubernetes supports multiple virtual clusters backed by the same physical cluster. These virtual clusters are called [namespaces](#). They let you partition resources into logically named groups.

Dashboard offers all available namespaces in a dropdown list, and allows you to create a new namespace. The namespace name may contain a maximum of 63 alphanumeric characters and dashes (-) but can not contain capital letters. Namespace names should not consist of only numbers. If the name is set as a number, such as 10, the pod will be put in the default namespace.

In case the creation of the namespace is successful, it is selected by default. If the creation fails, the first namespace is selected.

- **Image Pull Secret:** In case the specified Docker container image is private, it may require [pull secret](#) credentials.

Dashboard offers all available secrets in a dropdown list, and allows you to create a new secret. The secret name must follow the DNS domain name syntax, for example `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 manifests (YAML or JSON configuration files). The manifests use Kubernetes [API](#) resource schemas.

As an alternative to specifying application details in the deploy wizard, you can define your application in one or more manifests, and upload the files using Dashboard.

Using Dashboard

Following sections describe views of the Kubernetes Dashboard UI; what they provide and how can they be used.

Navigation

When there are Kubernetes objects defined in the cluster, Dashboard shows them in the initial view. By default only objects from the *default* namespace are shown and this can be changed using the namespace selector located in the navigation menu.

Dashboard shows most Kubernetes object kinds and groups them in a few menu categories.

Admin overview

For cluster and namespace administrators, Dashboard lists Nodes, Namespaces and PersistentVolumes and has detail views for them. Node list view contains CPU and memory usage metrics aggregated across all Nodes. The details view shows the metrics for a Node, its specification, status, allocated resources, events and pods running on the node.

Workloads

Shows all applications running in the selected namespace. The view lists applications by workload kind (for example: Deployments, ReplicaSets, StatefulSets). Each workload kind can be viewed separately. The lists summarize actionable information about the workloads, such as the number of ready pods for a ReplicaSet 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 ReplicaSet is controlling or new ReplicaSets and HorizontalPodAutoscalers for Deployments.

Services

Shows Kubernetes resources that allow for exposing services to external world and discovering them within a cluster. For that reason, Service and Ingress views show Pods targeted by them, internal endpoints for cluster connections and external endpoints for external users.

Storage

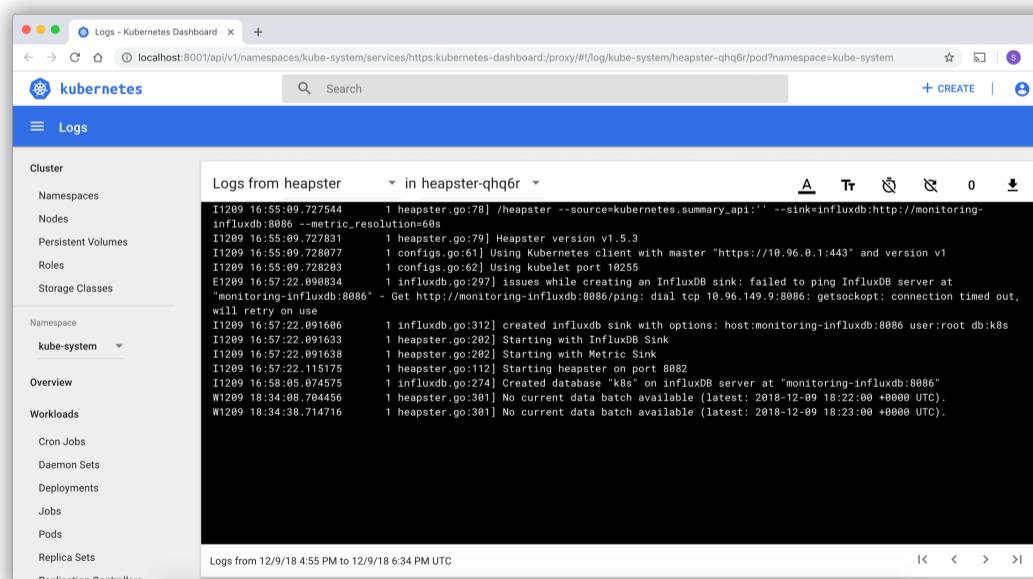
Storage view shows PersistentVolumeClaim resources which are used by applications for storing data.

ConfigMaps and Secrets

Shows all Kubernetes resources that are used for live configuration of applications running in clusters. The view allows for editing and managing config objects and displays secrets hidden by default.

Logs viewer

Pod lists and detail pages link to a logs viewer that is built into Dashboard. The viewer allows for drilling down logs from containers belonging to a single Pod.



What's next

For more information, see the [Kubernetes Dashboard project page](#).

10.2 - Accessing Clusters

This topic discusses multiple ways to interact with clusters.

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 set up 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 reference](#).

Directly accessing the REST API

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

- Run `kubectl` in proxy mode.
 - Recommended approach.
 - Uses stored `apiserver` location.
 - Verifies identity of `apiserver` using self-signed cert. No MITM possible.
 - Authenticates to `apiserver`.
 - In future, may do intelligent client-side load-balancing and failover.
- Provide the location and credentials directly to the http client.
 - Alternate approach.
 - Works with some types of client code that are confused by using a proxy.
 - Need to import a root cert into your browser to protect against MITM.

Using `kubectl proxy`

The following command runs `kubectl` in a mode where it acts as a reverse proxy. It handles locating the `apiserver` and authenticating. Run it like this:

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

See [kubectl proxy](#) for more details.

Then you can explore the API with `curl`, `wget`, or a browser, replacing `localhost` with `[::1]` for IPv6, like so:

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

The output is similar to this:

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

Without kubectl proxy

Use `kubectl apply` and `kubectl describe secret...` to create a token for the default service account with grep/cut:

First, create the Secret, requesting a token for the default ServiceAccount:

```
kubectl apply -f - <<EOF
apiVersion: v1
kind: Secret
metadata:
  name: default-token
  annotations:
    kubernetes.io/service-account.name: default
type: kubernetes.io/service-account-token
EOF
```

Next, wait for the token controller to populate the Secret with a token:

```
while ! kubectl describe secret default-token | grep -E '^token' >/dev/null
do
  echo "waiting for token..." >&2
  sleep 1
done
```

Capture and use the generated token:

```
APISERVER=$(kubectl config view --minify | grep server | cut -f 2- -d '/')
TOKEN=$(kubectl describe secret default-token | grep -E '^token' | cut -f 2 -d ' ')
curl $APISERVER/api --header "Authorization: Bearer $TOKEN" --insecure
```

The output is similar to this:

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

Using jsonpath :

```
APISERVER=$(kubectl config view --minify -o jsonpath='{.clusters[0].cluster.server}')
TOKEN=$(kubectl get secret default-token -o jsonpath='{.data.token}' | base64 --decode)
curl $APISERVER/api --header "Authorization: Bearer $TOKEN" --insecure
```

The output is similar to this:

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

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

On some clusters, the apiserver does not require authentication; it may serve on localhost, or be protected by a firewall. There is not a standard for this. [Controlling Access to the API](#) describes how a cluster admin can configure this.

Programmatic access to the API

Kubernetes officially supports [Go](#) and [Python](#) client libraries.

Go client

- To get the library, run the following command: `go get k8s.io/client-go@kubernetes-<kubernetes-version-number>`, see [INSTALL.md](#) for detailed installation instructions. See <https://github.com/kubernetes/client-go> to see which versions are supported.

- Write an application atop of the client-go clients. Note that client-go defines its own API objects, so if needed, please import API definitions from client-go rather than from the main repository, e.g., `import "k8s.io/client-go/kubernetes"` is correct.

The Go client can use the same [kubeconfig file](#) as the kubectl CLI does to locate and authenticate to the apiserver. See this [example](#).

If the application is deployed as a Pod in the cluster, please refer to the [next section](#).

Python client

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

The Python client can use the same [kubeconfig file](#) as the kubectl CLI does to locate and authenticate to the apiserver. See this [example](#).

Other languages

There are [client libraries](#) for accessing the API from other languages. See documentation for other libraries for how they authenticate.

Accessing the API from a Pod

When accessing the API from a pod, locating and authenticating to the API server are somewhat different.

Please check [Accessing the API from within a Pod](#) for more details.

Accessing services running on the cluster

The previous section describes how to connect to the Kubernetes API server. For information about connecting to other services running on a Kubernetes cluster, see [Access Cluster Services](#).

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 only 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 set up correctly.

10.3 - 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`.

Warning: Only use kubeconfig files from trusted sources. Using a specially-crafted kubeconfig file could result in malicious code execution or file exposure. If you must use an untrusted kubeconfig file, inspect it carefully first, much as you would a shell script.

Before you begin

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

- [KillerCoda](#)
- [Play with Kubernetes](#)

To check that `kubectl` is installed, run `kubectl version --client`. The `kubectl` version should be [within one minor version](#) of your cluster's API server.

Define clusters, users, and contexts

Suppose you have two clusters, one for development work and one for test 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 `test` 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 test 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: test

users:
- name: developer
- name: experimenter

contexts:
- context:
  name: dev-frontend
- context:
  name: dev-storage
- context:
  name: exp-test
```

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=http://127.0.0.1:8080
kubectl config --kubeconfig=config-demo set-cluster test --server=http://127.0.0.1:8081
```

Add user details to your configuration file:

Caution: Storing passwords in Kubernetes client config is risky. A better alternative would be to use a credential plugin and store them separately. See: [client-go credential plugins](#)

```
kubectl config --kubeconfig=config-demo set-credentials developer --client-key=development-key --client-cert=development-cert --password=development-password
kubectl config --kubeconfig=config-demo set-credentials experimenter --client-key=experimenter-key --client-cert=experimenter-cert --password=experimenter-password
```

Note:

- To delete a user you can run `kubectl --kubeconfig=config-demo config unset users.<name>`
- To remove a cluster, you can run `kubectl --kubeconfig=config-demo config unset clusters.<name>`
- To remove a context, you can run `kubectl --kubeconfig=config-demo config unset contexts.<name>`

Add context details to your configuration file:

```
kubectl config --kubeconfig=config-demo set-context dev-frontend --cluster development
kubectl config --kubeconfig=config-demo set-context dev-storage --cluster storage
kubectl config --kubeconfig=config-demo set-context exp-test --cluster test
```

Open your `config-demo` file to see the added details. As an alternative to opening the `config-demo` file, you can use the `config view` command.

```
kubectl config --kubeconfig=config-demo view
```

The output shows the two clusters, two users, and three contexts:

```
apiVersion: v1
clusters:
- cluster:
  certificate-authority: fake-ca-file
  server: https://1.2.3.4
  name: development
- cluster:
  insecure-skip-tls-verify: true
  server: https://5.6.7.8
  name: test
contexts:
- context:
  cluster: development
  namespace: frontend
  user: developer
  name: dev-frontend
- context:
  cluster: development
  namespace: storage
  user: developer
  name: dev-storage
- context:
  cluster: test
  namespace: default
  user: experimenter
  name: exp-test
current-context: ""
kind: Config
preferences: {}
users:
- name: developer
  user:
    client-certificate: fake-cert-file
    client-key: fake-key-file
- name: experimenter
  user:
    # Documentation note (this comment is NOT part of the command output)
    # Storing passwords in Kubernetes client config is risky.
    # A better alternative would be to use a credential plugin
    # and store the credentials separately.
    # See https://kubernetes.io/docs/reference/access-authn-authz/authenticating/
    password: some-password
    username: exp
```

The `fake-ca-file`, `fake-cert-file` and `fake-key-file` above are the placeholders for the pathnames of the certificate files. You need to change these to the actual pathnames of certificate files in your environment.

Sometimes you may want to use Base64-encoded data embedded here instead of separate certificate files; in that case you need to add the suffix `-data` to the keys, for example, `certificate-authority-data`, `client-certificate-data`, `client-key-data`.

Each context is a triple (cluster, user, namespace). For example, the `dev-frontend` context says, "Use the credentials of the `developer` user to access the `frontend` namespace of the `development` cluster".

Set the current context:

```
kubectl config --kubeconfig=config-demo use-context dev-frontend
```

Now whenever you enter a `kubectl` command, the action will apply to the cluster, and namespace listed in the `dev-frontend` context. And the command will use the credentials of the user listed in the `dev-frontend` context.

To see only the configuration information associated with the current context, use the `--minify` flag.

```
kubectl config --kubeconfig=config-demo view --minify
```

The output shows configuration information associated with the `dev-frontend` context:

```
apiVersion: v1
clusters:
- cluster:
  certificate-authority: fake-ca-file
  server: https://1.2.3.4
  name: development
contexts:
- context:
  cluster: development
  namespace: frontend
  user: developer
  name: dev-frontend
current-context: dev-frontend
kind: Config
preferences: {}
users:
- name: developer
  user:
    client-certificate: fake-cert-file
    client-key: fake-key-file
```

Now suppose you want to work for a while in the test cluster.

Change the current context to `exp-test`:

```
kubectl config --kubeconfig=config-demo use-context exp-test
```

Now any `kubectl` command you give will apply to the default namespace of the `test` cluster. And the command will use the credentials of the user listed in the `exp-test` context.

View configuration associated with the new current context, exp-test .

```
kubectl config --kubeconfig=config-demo view --minify
```

Finally, suppose you want to work for a while in the storage namespace of the development cluster.

Change the current context to dev-storage :

```
kubectl config --kubeconfig=config-demo use-context dev-storage
```

View configuration associated with the new current context, dev-storage .

```
kubectl config --kubeconfig=config-demo view --minify
```

Create a second configuration file

In your config-exercise directory, create a file named config-demo-2 with this content:

```
apiVersion: v1
kind: Config
preferences: {}

contexts:
- context:
  cluster: development
  namespace: ramp
  user: developer
  name: dev-ramp-up
```

The preceding configuration file defines a new context named dev-ramp-up .

Set the KUBECONFIG environment variable

See whether you have an environment variable named KUBECONFIG . If so, save the current value of your KUBECONFIG environment variable, so you can restore it later. For example:

Linux

```
export KUBECONFIG_SAVED="$KUBECONFIG"
```

Windows PowerShell

```
$Env:KUBECONFIG_SAVED=$Env:KUBECONFIG
```

The `KUBECONFIG` environment variable is a list of paths to configuration files. The list is colon-delimited for Linux and Mac, and semicolon-delimited for Windows. If you have a `KUBECONFIG` environment variable, familiarize yourself with the configuration files in the list.

Temporarily append two paths to your `KUBECONFIG` environment variable. For example:

Linux

```
export KUBECONFIG="${KUBECONFIG}:config-demo:config-demo-2"
```

Windows PowerShell

```
$Env:KUBECONFIG=("config-demo;config-demo-2")
```

In your `config-exercise` directory, enter this command:

```
kubectl config view
```

The output shows merged information from all the files listed in your `KUBECONFIG` environment variable. In particular, notice that the merged information has the `dev-ramp-up` context from the `config-demo-2` file and the three contexts from the `config-demo` file:

```
contexts:
- context:
  cluster: development
  namespace: frontend
  user: developer
  name: dev-frontend
- context:
  cluster: development
  namespace: ramp
  user: developer
  name: dev-ramp-up
- context:
  cluster: development
  namespace: storage
  user: developer
  name: dev-storage
- context:
  cluster: test
  namespace: default
  user: experimenter
  name: exp-test
```

For more information about how kubeconfig files are merged, see [Organizing Cluster Access Using kubeconfig Files](#)

Explore the \$HOME/.kube directory

If you already have a cluster, and you can use `kubectl` to interact with the cluster, then you probably have a file named `config` in the `$HOME/.kube` directory.

Go to `$HOME/.kube`, and see what files are there. Typically, there is a file named `config`. There might also be other configuration files in this directory. Briefly familiarize yourself with the contents of these files.

Append \$HOME/.kube/config to your KUBECONFIG environment variable

If you have a `$HOME/.kube/config` file, and it's not already listed in your `KUBECONFIG` environment variable, append it to your `KUBECONFIG` environment variable now. For example:

Linux

```
export KUBECONFIG="${KUBECONFIG}:${HOME}/.kube/config"
```

Windows Powershell

```
$Env:KUBECONFIG="$Env:KUBECONFIG;$HOME\.kube\config"
```

View configuration information merged from all the files that are now listed in your `KUBECONFIG` environment variable. In your config-exercise directory, enter:

```
kubectl config view
```

Clean up

Return your `KUBECONFIG` environment variable to its original value. For example:

Linux

```
export KUBECONFIG="$KUBECONFIG_SAVED"
```

Windows PowerShell

```
$Env:KUBECONFIG=$Env:KUBECONFIG_SAVED
```

Check the subject represented by the kubeconfig

It is not always obvious what attributes (username, groups) you will get after authenticating to the cluster. It can be even more challenging if you are managing more than one cluster at the same time.

There is a `kubectl` subcommand to check subject attributes, such as `username`, for your selected Kubernetes client context: `kubectl auth whoami`.

Read [API access to authentication information for a client](#) to learn about this in more detail.

What's next

- [Organizing Cluster Access Using kubeconfig Files](#)
- [kubectl config](#)

10.4 - Use Port Forwarding to Access Applications in a Cluster

This page shows how to use `kubectl port-forward` to connect to a MongoDB server running in a Kubernetes cluster. This type of connection can be useful for database debugging.

Before you begin

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

- [Killercoda](#)
- [Play with Kubernetes](#)

Your Kubernetes server must be at or later than version v1.10. To check the version, enter `kubectl version`.

- Install [MongoDB Shell](#).

Creating MongoDB deployment and service

1. Create a Deployment that runs MongoDB:

```
kubectl apply -f https://k8s.io/examples/application/mongodb/mong
```

The output of a successful command verifies that the deployment was created:

```
deployment.apps/mongo created
```

View the pod status to check that it is ready:

```
kubectl get pods
```

The output displays the pod created:

NAME	READY	STATUS	RESTARTS	AGE
mongo-75f59d57f4-4nd6q	1/1	Running	0	2m4s

View the Deployment's status:

```
kubectl get deployment
```

The output displays that the Deployment was created:

NAME	READY	UP-TO-DATE	AVAILABLE	AGE
mongo	1/1	1	1	2m21s

The Deployment automatically manages a ReplicaSet. View the ReplicaSet status using:

```
kubectl get replicaset
```

The output displays that the ReplicaSet was created:

NAME	DESIRED	CURRENT	READY	AGE
mongo-75f59d57f4	1	1	1	3m12s

2. Create a Service to expose MongoDB on the network:

```
kubectl apply -f https://k8s.io/examples/application/mongodb/mong
```

The output of a successful command verifies that the Service was created:

```
service/mongo created
```

Check the Service created:

```
kubectl get service mongo
```

The output displays the service created:

NAME	TYPE	CLUSTER-IP	EXTERNAL-IP	PORT(S)	AGE
mongo	ClusterIP	10.96.41.183	<none>	27017/TCP	11s

3. Verify that the MongoDB server is running in the Pod, and listening on port 27017:

```
# Change mongo-75f59d57f4-4nd6q to the name of the Pod
kubectl get pod mongo-75f59d57f4-4nd6q --template='{{(index $(
```

The output displays the port for MongoDB in that Pod:

```
27017
```

27017 is the TCP port allocated to MongoDB on the internet.

Forward a local port to a port on the Pod

1. `kubectl port-forward` allows using resource name, such as a pod name, to select a matching pod to port forward to.

```
# Change mongo-75f59d57f4-4nd6q to the name of the Pod
kubectl port-forward mongo-75f59d57f4-4nd6q 28015:27017
```

which is the same as

```
kubectl port-forward pods/mongo-75f59d57f4-4nd6q 28015:27017
```

or

```
kubectl port-forward deployment/mongo 28015:27017
```

or

```
kubectl port-forward replicaset/mongo-75f59d57f4 28015:27017
```

or

```
kubectl port-forward service/mongo 28015:27017
```

Any of the above commands works. The output is similar to this:

```
Forwarding from 127.0.0.1:28015 -> 27017
Forwarding from [::1]:28015 -> 27017
```

Note: `kubectl port-forward` does not return. To continue with the exercises, you will need to open another terminal.

2. Start the MongoDB command line interface:

```
mongosh --port 28015
```

3. At the MongoDB command line prompt, enter the `ping` command:

```
db.runCommand( { ping: 1 } )
```

A successful ping request returns:

```
{ ok: 1 }
```

Optionally let `kubectl` choose the local port

If you don't need a specific local port, you can let `kubectl` choose and allocate the local port and thus relieve you from having to manage local port conflicts, with the slightly simpler syntax:

```
kubectl port-forward deployment/mongo :27017
```

The `kubectl` tool finds a local port number that is not in use (avoiding low ports numbers, because these might be used by other applications). The output is similar to:

```
Forwarding from 127.0.0.1:63753 -> 27017
Forwarding from [::1]:63753 -> 27017
```

Discussion

Connections made to local port 28015 are forwarded to port 27017 of the Pod that is running the MongoDB server. With this connection in place, you can use your local workstation to debug the database that is running in the Pod.

Note: `kubectl port-forward` is implemented for TCP ports only. The support for UDP protocol is tracked in [issue 47862](#).

What's next

Learn more about [kubectl port-forward](#).

10.5 - 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.

Before you begin

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

- [Killercoda](#)
- [Play with Kubernetes](#)

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.

Creating a service for an application running in two pods

Here is the configuration file for the application Deployment:

```
service/access/hello-application.yaml 
```

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: hello-world
spec:
  selector:
    matchLabels:
      run: load-balancer-example
  replicas: 2
  template:
    metadata:
      labels:
        run: load-balancer-example
    spec:
      containers:
        - name: hello-world
          image: gcr.io/google-samples/node-hello:1.0
          ports:
            - containerPort: 8080
              protocol: TCP
```

1. Run a Hello World application in your cluster: Create the application

Deployment using the file above:

```
kubectl apply -f https://k8s.io/examples/service/access/hello-app
```

The preceding command creates a Deployment and an associated ReplicaSet. 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=exam
```

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=
```

The output is similar to this:

NAME	READY	STATUS	...	IP
hello-world-2895499144-bsbk5	1/1	Running	...	10.200.1.4
hello-world-2895499144-m1pwt	1/1	Running	...	10.200.2.5

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

Follow the [Connecting Applications with Services](#) tutorial.

10.6 - Connect a Frontend to a Backend Using Services

This task shows how to create a *frontend* and a *backend* microservice. The backend microservice is a hello greeter. The frontend exposes the backend using nginx and a Kubernetes Service object.

Objectives

- Create and run a sample `hello` backend microservice using a [Deployment](#) object.
- Use a Service object to send traffic to the backend microservice's multiple replicas.
- Create and run a `nginx` frontend microservice, also using a [Deployment](#) object.
- Configure the frontend microservice to send traffic to the backend microservice.
- Use a Service object of `type=LoadBalancer` to expose the frontend microservice outside 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. It is recommended to run this tutorial on a cluster with at least two nodes that are not acting as control plane hosts. If you do not already have a cluster, you can create one by using [minikube](#) or you can use one of these Kubernetes playgrounds:

- [Killercoda](#)
- [Play with Kubernetes](#)

To check the version, enter `kubectl version`.

This task uses [Services with external load balancers](#), which require a supported environment. If your environment does not support this, you can use a Service of type [NodePort](#) instead.

Creating the backend using a Deployment

The backend is a simple hello greeter microservice. Here is the configuration file for the backend Deployment:

[service/access/backend-deployment.yaml](#) 

```
---  
apiVersion: apps/v1  
kind: Deployment  
metadata:  
  name: backend  
spec:  
  selector:  
    matchLabels:  
      app: hello  
      tier: backend  
      track: stable  
  replicas: 3  
  template:  
    metadata:  
      labels:  
        app: hello  
        tier: backend  
        track: stable  
    spec:  
      containers:  
        - name: hello  
          image: "gcr.io/google-samples/hello-go-gke:1.0"  
          ports:  
            - name: http  
              containerPort: 80  
...  
...
```

Create the backend Deployment:

```
kubectl apply -f https://k8s.io/examples/service/access/backend-deploy
```

View information about the backend Deployment:

```
kubectl describe deployment backend
```

The output is similar to this:

```
Name: backend
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: 3 desired | 3 updated | 3 total | 3 available
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 (3/3 replicas created)
Events:
  ...
```

Creating the `hello` Service object

The key to sending requests from a frontend to a backend is the backend Service. A Service creates a persistent IP address and DNS name entry so that the backend microservice can always be reached. A Service uses selectors to find the Pods that it routes traffic to.

First, explore the Service configuration file:

[service/access/backend-service.yaml](#) 

```
...
apiVersion: v1
kind: Service
metadata:
  name: hello
spec:
  selector:
    app: hello
    tier: backend
  ports:
    - protocol: TCP
      port: 80
      targetPort: http
...
```

In the configuration file, you can see that the Service, named `hello` routes traffic to Pods that have the labels `app: hello` and `tier: backend`.

Create the backend Service:

```
kubectl apply -f https://k8s.io/examples/service/access/backend-service.yaml
```

At this point, you have a `backend` Deployment running three replicas of your `hello` application, and you have a Service that can route traffic to them. However, this service is neither available nor resolvable outside the cluster.

Creating the frontend

Now that you have your backend running, you can create a frontend that is accessible outside the cluster, and connects to the backend by proxying requests to it.

The frontend sends requests 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 `examples/service/access/backend-service.yaml` configuration file.

The Pods in the frontend Deployment run a `nginx` image that is configured to proxy requests to the `hello` backend Service. Here is the `nginx` configuration file:

[service/access/frontend-nginx.conf](#)

```
# The identifier Backend is internal to nginx, and used to name this upstream
upstream Backend {
    # hello is the internal DNS name used by the backend Service inside
    server hello;
}
server {
    listen 80;
```

```
    location / {
        # The following statement will proxy traffic to the upstream named Backend
        proxy_pass http://Backend;
    }
}
```

Similar to the backend, the frontend has a Deployment and a Service. An important difference to notice between the backend and frontend services, is that the configuration for the frontend Service has `type: LoadBalancer`, which means that the Service uses a load balancer provisioned by your cloud provider and will be accessible from outside the cluster.

[service/access/frontend-service.yaml](#)

```
---  
apiVersion: v1  
kind: Service  
metadata:  
  name: frontend  
spec:  
  selector:  
    app: hello  
    tier: frontend  
  ports:  
  - protocol: "TCP"  
    port: 80  
    targetPort: 80  
  type: LoadBalancer  
...
```

[service/access/frontend-deployment.yaml](#)

```
---  
apiVersion: apps/v1  
kind: Deployment  
metadata:  
  name: frontend  
spec:  
  selector:  
    matchLabels:  
      app: hello  
      tier: frontend  
      track: stable  
  replicas: 1  
  template:  
    metadata:  
      labels:  
        app: hello  
        tier: frontend  
        track: stable  
    spec:  
      containers:  
      - name: nginx  
        image: "gcr.io/google-samples/hello-frontend:1.0"  
      lifecycle:  
        preStop:  
          exec:  
            command: ["/usr/sbin/nginx", "-s", "quit"]  
...
```

Create the frontend Deployment and Service:

```
kubectl apply -f https://k8s.io/examples/service/access/frontend-deploy  
kubectl apply -f https://k8s.io/examples/service/access/frontend-service
```

The output verifies that both resources were created:

```
deployment.apps/frontend created  
service/frontend created
```

Note: The nginx configuration is baked into the [container image](#). A better way to do this would be to use a [ConfigMap](#), so that you can change the configuration more easily.

Interact with the frontend Service

Once you've created a Service of type LoadBalancer, you can use this command to find the external IP:

```
kubectl get service frontend --watch
```

This displays the configuration for the `frontend` Service and watches for changes. Initially, the external IP is listed as `<pending>`:

NAME	TYPE	CLUSTER-IP	EXTERNAL-IP	PORT(S)	AGE
frontend	LoadBalancer	10.51.252.116	<pending>	80/TCP	10s

As soon as an external IP is provisioned, however, the configuration updates to include the new IP under the `EXTERNAL-IP` heading:

NAME	TYPE	CLUSTER-IP	EXTERNAL-IP	PORT(S)	AGE
frontend	LoadBalancer	10.51.252.116	XXX.XXX.XXX.XXX	80/TCP	1m

That IP can now be used to interact with the `frontend` service from outside the cluster.

Send traffic through the frontend

The frontend and backend are now connected. You can hit the endpoint by using the curl command on the external IP of your frontend Service.

```
curl http://${EXTERNAL_IP} # replace this with the EXTERNAL-IP you saw
```

The output shows the message generated by the backend:

```
{"message": "Hello"}
```

Cleaning up

To delete the Services, enter this command:

```
kubectl delete services frontend backend
```

To delete the Deployments, the ReplicaSets and the Pods that are running the backend and frontend applications, enter this command:

```
kubectl delete deployment frontend backend
```

What's next

- Learn more about [Services](#)
- Learn more about [ConfigMaps](#)
- Learn more about [DNS for Service and Pods](#)

10.7 - 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 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*.

You can also use an [Ingress](#) in place of Service. For more information, check the [Ingress](#) documentation.

Before you begin

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

- [Killercoda](#)
- [Play with Kubernetes](#)

Your cluster must be running in a cloud or other environment that already has support for configuring external load balancers.

Create a Service

Create a Service from a manifest

To create an external load balancer, add the following line to your Service manifest:

```
type: LoadBalancer
```

Your manifest might then look like:

```
apiVersion: v1
kind: Service
metadata:
  name: example-service
spec:
  selector:
    app: example
  ports:
    - port: 8765
      targetPort: 9376
  type: LoadBalancer
```

Create a Service using kubectl

You can alternatively create the service with the `kubectl expose` command and its `--type=LoadBalancer` flag:

```
kubectl expose deployment 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 [Deployment](#) 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 similar to:

```
Name:           example-service
Namespace:      default
Labels:          app=example
Annotations:    <none>
Selector:        app=example
Type:           LoadBalancer
IP Families:   <none>
IP:             10.3.22.96
IPs:            10.3.22.96
LoadBalancer Ingress: 192.0.2.89
Port:           <unset>  8765/TCP
TargetPort:     9376/TCP
NodePort:       <unset>  30593/TCP
Endpoints:     172.17.0.3:9376
Session Affinity: None
External Traffic Policy: Cluster
Events:         <none>
```

The load balancer's 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

By default, the source IP seen in the target container is *not the original source IP* of the client. To enable preservation of the client IP, the following fields can be configured in the `.spec` of the 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.

- `.spec.healthCheckNodePort` - specifies the health check node port (numeric port number) for the service. If you don't specify `healthCheckNodePort`, the service controller allocates a port from your cluster's NodePort range.

You can configure that range by setting an API server command line option, `--service-node-port-range`. The Service will use the user-specified `healthCheckNodePort` value if you specify it, provided that the `Service` type is set to LoadBalancer and `externalTrafficPolicy` is set to `Local`.

Setting `externalTrafficPolicy` to Local in the Service manifest activates this feature. For example:

```
apiVersion: v1
kind: Service
metadata:
  name: example-service
spec:
  selector:
    app: example
  ports:
    - port: 8765
      targetPort: 9376
  externalTrafficPolicy: Local
  type: LoadBalancer
```

Caveats and limitations when preserving source IPs

Load balancing services from some cloud providers do not let you configure different weights for each target.

With each target weighted equally in terms of sending traffic to Nodes, external traffic is not equally load balanced across different Pods. The external load balancer is unaware of the number of Pods on each node that are used as a target.

Where `NumServicePods << _NumNodes OR NumServicePods >> NumNodes`, a fairly close-to-equal distribution will be seen, even without weights.

Internal pod to pod traffic should behave similar to ClusterIP services, with equal probability across all pods.

Garbage collecting load balancers

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

In usual case, the correlating load balancer resources in cloud provider should be cleaned up soon after a LoadBalancer type Service is deleted. But it is known that there are various corner cases where cloud resources are orphaned after the associated Service is deleted. Finalizer Protection for Service LoadBalancers was introduced to prevent this from happening. By using finalizers, a Service resource will never be deleted until the correlating load balancer resources are also deleted.

Specifically, if a Service has `type LoadBalancer`, the service controller will attach a finalizer named `service.kubernetes.io/load-balancer-cleanup`. The finalizer will only be removed after the load balancer resource is cleaned

up. This prevents dangling load balancer resources even in corner cases such as the service controller crashing.

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` equals `ClusterIP` to pods within the cluster and extends it by programming the (external to Kubernetes) load balancer with entries for the nodes hosting the relevant Kubernetes pods. The Kubernetes control plane automates the creation of the external load balancer, health checks (if needed), and packet filtering rules (if needed). Once the cloud provider allocates an IP address for the load balancer, the control plane looks up that external IP address and populates it into the `Service` object.

What's next

- Follow the [Connecting Applications with Services](#) tutorial
- Read about [Service](#)
- Read about [Ingress](#)

10.8 - 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

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

- [Killercoda](#)
- [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 Container images 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={.items[*].spec['initContainers', 'containers'][*].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=".items[*].spec['initContainers', 'containers'][*].image" | tr -s '[:space:]' '\n' | sort | uniq -c
```

The jsonpath is interpreted as follows:

- `.items[*]` : for each returned value
- `.spec` : get the spec
- `['initContainers', 'containers'][*]` : for each container
- `.image` : get the image

Note: When fetching a single Pod by name, for example `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 Container images 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"}-  
sort
```

List Container images 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=".items[*].spec.contain
```

List Container images 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=".items[*].spec.
```

List Container images 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 .
```

What's next

Reference

- [Jsonpath](#) reference guide
- [Go template](#) reference guide

10.9 - Set up Ingress on Minikube with the NGINX Ingress Controller

An [Ingress](#) is an API object that defines rules which allow external access to services in a cluster. An [Ingress controller](#) fulfills the rules set in the Ingress.

This page shows you how to set up a simple Ingress which routes requests to Service 'web' or 'web2' depending on the HTTP URI.

Before you begin

This tutorial assumes that you are using `minikube` to run a local Kubernetes cluster. Visit [Install tools](#) to learn how to install `minikube`.

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

- [Killercode](#)
- [Play with Kubernetes](#)

Your Kubernetes server must be at or later than version 1.19. To check the version, enter `kubectl version`. If you are using an older Kubernetes version, switch to the documentation for that version.

Create a minikube cluster

If you haven't already set up a cluster locally, run `minikube start` to create a cluster.

Enable the Ingress controller

1. To enable the NGINX Ingress controller, run the following command:

```
minikube addons enable ingress
```

2. Verify that the NGINX Ingress controller is running

```
kubectl get pods -n ingress-nginx
```

Note: It can take up to a minute before you see these pods running OK.

The output is similar to:

NAME	READY	STATUS	REPLICAS
ingress-nginx-admission-create-g9g49	0/1	Completed	0
ingress-nginx-admission-patch-rqp78	0/1	Completed	1
ingress-nginx-controller-59b45fb494-26npt	1/1	Running	0

Deploy a hello, world app

1. Create a Deployment using the following command:

```
kubectl create deployment web --image=gcr.io/google-samples/hello
```

The output should be:

```
deployment.apps/web created
```

2. Expose the Deployment:

```
kubectl expose deployment web --type=NodePort --port=8080
```

The output should be:

```
service/web exposed
```

3. Verify the Service is created and is available on a node port:

```
kubectl get service web
```

The output is similar to:

NAME	TYPE	CLUSTER-IP	EXTERNAL-IP	PORT(S)
web	NodePort	10.104.133.249	<none>	8080:31637/TCP

4. Visit the Service via NodePort:

```
minikube service web --url
```

The output is similar to:

```
http://172.17.0.15:31637
```

```
curl http://172.17.0.15:31637
```

The output is similar to:

```
Hello, world!  
Version: 1.0.0  
Hostname: web-55b8c6998d-8k564
```

You can now access the sample application via the Minikube IP address and NodePort. The next step lets you access the application using the Ingress resource.

Create an Ingress

The following manifest defines an Ingress that sends traffic to your Service via `hello-world.info`.

1. Create `example-ingress.yaml` from the following file:

[service/networking/example-ingress.yaml](#) 

```
apiVersion: networking.k8s.io/v1  
kind: Ingress  
metadata:  
  name: example-ingress  
  annotations:  
    nginx.ingress.kubernetes.io/rewrite-target: /$1  
spec:  
  rules:  
    - host: hello-world.info  
      http:  
        paths:  
          - path: /  
            pathType: Prefix  
            backend:  
              service:  
                name: web  
                port:  
                  number: 8080
```

2. Create the Ingress object by running the following command:

```
kubectl apply -f https://k8s.io/examples/service/networking/examp
```

The output should be:

```
ingress.networking.k8s.io/example-ingress created
```

3. Verify the IP address is set:

```
kubectl get ingress
```

Note: This can take a couple of minutes.

You should see an IPv4 address in the `ADDRESS` column; for example:

NAME	CLASS	HOSTS	ADDRESS	PORTS
example-ingress	<none>	hello-world.info	172.17.0.15	80

4. Verify that the Ingress controller is directing traffic:

```
curl --resolve "hello-world.info:80:$(`minikube ip`)" -i http://h
```

You should see:

```
Hello, world!
Version: 1.0.0
Hostname: web-55b8c6998d-8k564
```

You can also visit `hello-world.info` from your browser.

- o **Optionally** Look up the external IP address as reported by `minikube`:

```
minikube ip
```

Add line similar to the following one to the bottom of the `/etc/hosts` file on your computer (you will need administrator access):

```
172.17.0.15 hello-world.info
```

Note: Change the IP address to match the output from `minikube ip`.

After you make this change, your web browser sends requests for `hello-world.info` URLs to Minikube.

Create a second Deployment

1. Create another Deployment using the following command:

```
kubectl create deployment web2 --image=gcr.io/google-samples/hell
```

The output should be:

```
deployment.apps/web2 created
```

2. Expose the second Deployment:

```
kubectl expose deployment web2 --port=8080 --type=NodePort
```

The output should be:

```
service/web2 exposed
```

Edit the existing Ingress

1. Edit the existing `example-ingress.yaml` manifest, and add the following lines at the end:

```
- path: /v2
  pathType: Prefix
  backend:
    service:
      name: web2
      port:
        number: 8080
```

2. Apply the changes:

```
kubectl apply -f example-ingress.yaml
```

You should see:

```
ingress.networking/example-ingress configured
```

Test your Ingress

1. Access the 1st version of the Hello World app.

```
curl --resolve "hello-world.info:80:$( minikube ip )" -i http://h
```

The output is similar to:

```
Hello, world!
Version: 1.0.0
Hostname: web-55b8c6998d-8k564
```

2. Access the 2nd version of the Hello World app.

```
curl --resolve "hello-world.info:80:$( minikube ip )" -i http://h
```

The output is similar to:

```
Hello, world!
Version: 2.0.0
Hostname: web2-75cd47646f-t8cjk
```

Note: If you did the optional step to update `/etc/hosts`, you can also visit [hello-world.info](#) and [hello-world.info/v2](#) from your browser.

What's next

- Read more about [Ingress](#)
- Read more about [Ingress Controllers](#)
- Read more about [Services](#)

10.10 - Communicate Between Containers in the Same Pod Using a Shared Volume

This page shows how to use a Volume to communicate between two Containers running in the same Pod. See also how to allow processes to communicate by [sharing process namespace](#) between containers.

Before you begin

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

- [Killercoda](#)
- [Play with Kubernetes](#)

To check the version, enter `kubectl version`.

Creating a Pod that runs two Containers

In this exercise, you create a Pod that runs two Containers. The two containers share a Volume that they can use to communicate. Here is the configuration file for the Pod:

[pods/two-container-pod.yaml](#) 

```
apiVersion: v1
kind: Pod
metadata:
  name: two-containers
spec:

  restartPolicy: Never

  volumes:
  - name: shared-data
    emptyDir: {}

  containers:
  - name: nginx-container
    image: nginx
    volumeMounts:
    - name: shared-data
      mountPath: /usr/share/nginx/html

  - name: debian-container
    image: debian
    volumeMounts:
    - name: shared-data
      mountPath: /pod-data
    command: ["/bin/sh"]
    args: ["-c", "echo Hello from the debian container > /pod-data/index.html"]
```

In the configuration file, you can see that the Pod has a Volume named `shared-data`.

The first container listed in the configuration file runs an nginx server. The mount path for the shared Volume is `/usr/share/nginx/html`. The second container is based on the debian image, and has a mount path of `/pod-data`. The second container runs the following command and then terminates.

```
echo Hello from the debian container > /pod-data/index.html
```

Notice that the second container writes the `index.html` file in the root directory of the nginx server.

Create the Pod and the two Containers:

```
kubectl apply -f https://k8s.io/examples/pods/two-container-pod.yaml
```

View information about the Pod and the Containers:

```
kubectl get pod two-containers --output=yaml
```

Here is a portion of the output:

```
apiVersion: v1
kind: Pod
metadata:
  ...
  name: two-containers
  namespace: default
  ...
spec:
  ...
  containerStatuses:

  - containerID: docker://c1d8abd1 ...
    image: debian
    ...
    lastState:
      terminated:
        ...
    name: debian-container
    ...

  - containerID: docker://96c1ff2c5bb ...
    image: nginx
    ...
    name: nginx-container
    ...
    state:
      running:
        ...

```

You can see that the debian Container has terminated, and the nginx Container is still running.

Get a shell to nginx Container:

```
kubectl exec -it two-containers -c nginx-container -- /bin/bash
```

In your shell, verify that nginx is running:

```
root@two-containers:/# apt-get update
root@two-containers:/# apt-get install curl procps
root@two-containers:/# ps aux
```

The output is similar to this:

USER	PID	...	STAT	START	TIME	COMMAND
root	1	...	Ss	21:12	0:00	nginx: master process nginx -g c

Recall that the debian Container created the `index.html` file in the nginx root directory. Use `curl` to send a GET request to the nginx server:

```
root@two-containers:/# curl localhost
```

The output shows that nginx serves a web page written by the debian container:

```
Hello from the debian container
```

Discussion

The primary reason that Pods can have multiple containers is to support helper applications that assist a primary application. Typical examples of helper applications are data pullers, data pushers, and proxies. Helper and primary applications often need to communicate with each other. Typically this is done through a shared filesystem, as shown in this exercise, or through the loopback network interface, localhost. An example of this pattern is a web server along with a helper program that polls a Git repository for new updates.

The Volume in this exercise provides a way for Containers to communicate during the life of the Pod. If the Pod is deleted and recreated, any data stored in the shared Volume is lost.

What's next

- Learn more about [patterns for composite containers](#).
- Learn about [composite containers for modular architecture](#).
- See [Configuring a Pod to Use a Volume for Storage](#).
- See [Configure a Pod to share process namespace between containers in a Pod](#)
- See [Volume](#).
- See [Pod](#).

10.11 - Configure DNS for a Cluster

Kubernetes offers a DNS cluster addon, which most of the supported environments enable by default. In Kubernetes version 1.11 and later, CoreDNS is recommended and is installed by default with kubeadm.

For more information on how to configure CoreDNS for a Kubernetes cluster, see the [Customizing DNS Service](#). An example demonstrating how to use Kubernetes DNS with kube-dns, see the [Kubernetes DNS sample plugin](#).

10.12 - Access Services Running on Clusters

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

Before you begin

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

- [Killercoda](#)
- [Play with Kubernetes](#)

To check the version, enter `kubectl version`.

Accessing services running on the cluster

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

Ways to connect

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

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

- Some clusters may allow you to ssh to a node in the cluster.
From there you may be able to access cluster services. This is a non-standard method, and will work on some clusters but not others. Browsers and other tools may or may not be installed.
Cluster DNS may not work.

Discovering builtin services

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

```
kubectl cluster-info
```

The output is similar to this:

```
Kubernetes master is running at https://192.0.2.1
elasticsearch-logging is running at https://192.0.2.1/api/v1/namespaces
kibana-logging is running at https://192.0.2.1/api/v1/namespaces/kube-s
kube-dns is running at https://192.0.2.1/api/v1/namespaces/kube-system/
grafana is running at https://192.0.2.1/api/v1/namespaces/kube-system/s
heapster is running at https://192.0.2.1/api/v1/namespaces/kube-system/
```

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://192.0.2.1/api/v1/namespaces/kube-system/services/elasticsearch-logging/proxy/` if suitable credentials are passed, or through a kubectl proxy at, for example:

```
http://localhost:8080/api/v1/namespaces/kube-
system/services/elasticsearch-logging/proxy/ .
```

Note: See [Access Clusters Using the Kubernetes API](#) for how to pass credentials or use kubectl proxy.

Manually constructing apiserver proxy URLs

As mentioned above, you use the `kubectl cluster-info` command to retrieve the service's proxy URL. To create proxy URLs that include service endpoints, suffixes, and parameters, you append to the service's proxy URL:

```
http://<kubernetes_master_address>/api/v1/namespaces/<namespace_name>/se
rvices/<service_name>[:<port_name>]/proxy
```

If you haven't specified a name for your port, you don't have to specify `<port_name>` in the URL. You can also use the port number in place of the `<port_name>` for both named and unnamed ports.

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>/serv
ices/<service_name>/proxy
```

The supported formats for the `<service_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 name or port number 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 name or port number using https

Examples

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

```
http://192.0.2.1/api/v1/namespaces/kube-system/services/elasticsearch/_sear
```

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

```
https://192.0.2.1/api/v1/namespaces/kube-system/services/elasticsearch/_clust
```

The health information is similar to this:

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

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

```
https://192.0.2.1/api/v1/namespaces/kube-system/services/https:elasticsearch/_clust
```

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.

11 - Extend Kubernetes

Understand advanced ways to adapt your Kubernetes cluster to the needs of your work environment.

11.1 - 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

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

- [Killercoda](#)
- [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 Kubernetes general CA.

Caution: Reusing the same CA for different client types can negatively impact the cluster's ability to function. For more information, see [CA Reusage and Conflicts](#).

Authentication Flow

Unlike Custom Resource Definitions (CRDs), the Aggregation API involves another server - your Extension apiserver - in addition to the standard Kubernetes apiserver. The Kubernetes apiserver will need to communicate with your extension apiserver, and your extension apiserver will need to communicate with the Kubernetes apiserver. In order for this communication to be secured, the Kubernetes apiserver uses x509 certificates to authenticate itself to the extension apiserver.

This section describes how the authentication and authorization flows work, and how to configure them.

The high-level flow is as follows:

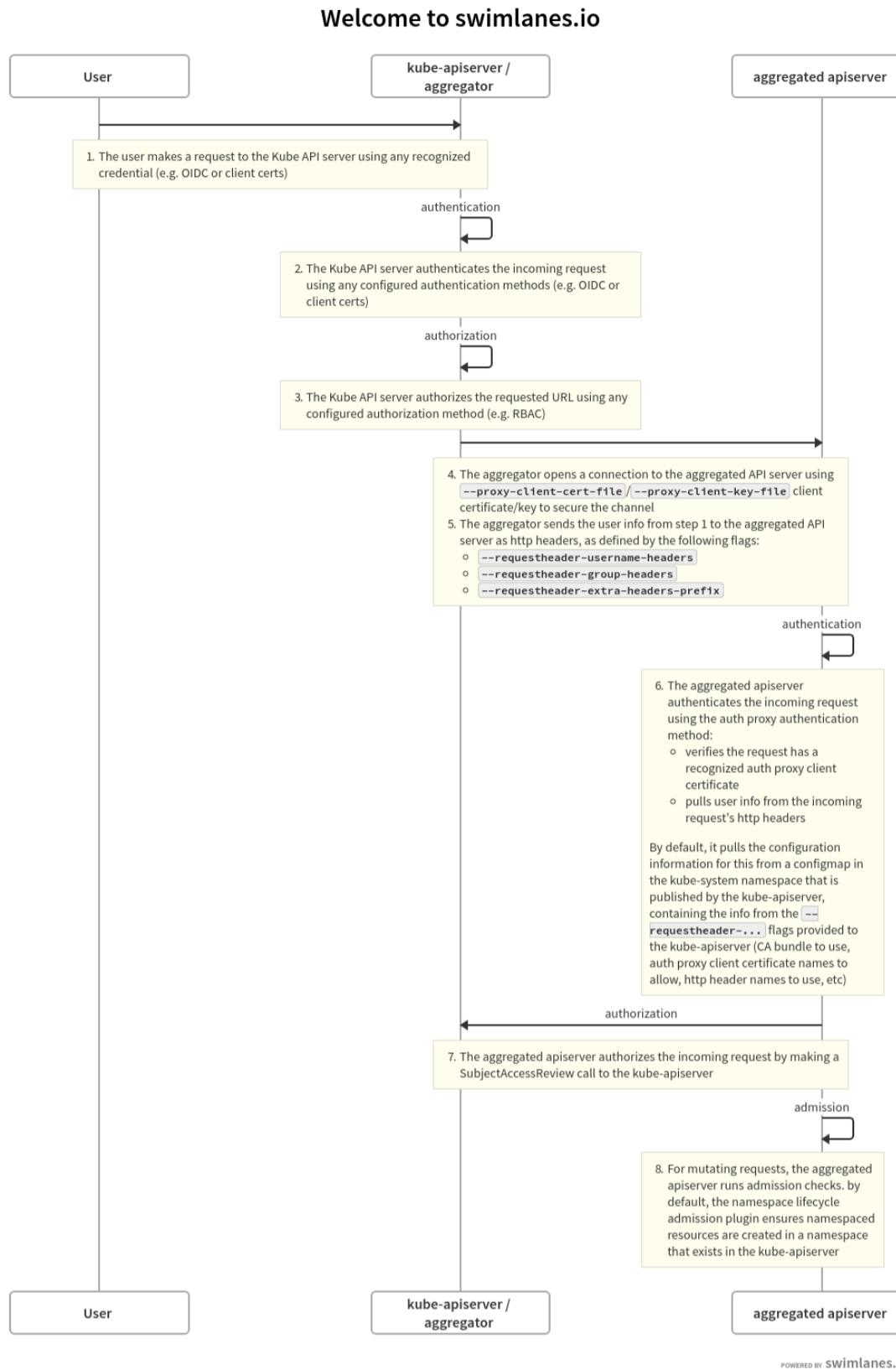
1. Kubernetes apiserver: authenticate the requesting user and authorize their rights to the requested API path.
2. Kubernetes apiserver: proxy the request to the extension apiserver
3. Extension apiserver: authenticate the request from the Kubernetes apiserver

4. Extension apiserver: authorize the request from the original user

5. Extension apiserver: execute

The rest of this section describes these steps in detail.

The flow can be seen in the following diagram.



The source for the above swimlanes can be found in the source of this document.

Kubernetes Apiserver Authentication and Authorization

A request to an API path that is served by an extension apiserver begins the same way as all API requests: communication to the Kubernetes apiserver. This path already has been registered with the Kubernetes apiserver by the extension apiserver.

The user communicates with the Kubernetes apiserver, requesting access to the path. The Kubernetes apiserver uses standard authentication and authorization configured with the Kubernetes apiserver to authenticate the user and authorize access to the specific path.

For an overview of authenticating to a Kubernetes cluster, see ["Authenticating to a Cluster"](#). For an overview of authorization of access to Kubernetes cluster resources, see ["Authorization Overview"](#).

Everything to this point has been standard Kubernetes API requests, authentication and authorization.

The Kubernetes apiserver now is prepared to send the request to the extension apiserver.

Kubernetes Apiserver Proxies the Request

The Kubernetes apiserver now will send, or proxy, the request to the extension apiserver that registered to handle the request. In order to do so, it needs to know several things:

1. How should the Kubernetes apiserver authenticate to the extension apiserver, informing the extension apiserver that the request, which comes over the network, is coming from a valid Kubernetes apiserver?
2. How should the Kubernetes apiserver inform the extension apiserver of the username and group for which the original request was authenticated?

In order to provide for these two, you must configure the Kubernetes apiserver using several flags.

Kubernetes Apiserver Client Authentication

The Kubernetes apiserver connects to the extension apiserver over TLS, authenticating itself using a client certificate. You must provide the following to the Kubernetes apiserver upon startup, using the provided flags:

- private key file via `--proxy-client-key-file`
- signed client certificate file via `--proxy-client-cert-file`
- certificate of the CA that signed the client certificate file via `--requestheader-client-ca-file`
- valid Common Name values (CNs) in the signed client certificate via `--requestheader-allowed-names`

The Kubernetes apiserver will use the files indicated by `--proxy-client-*file` to authenticate to the extension apiserver. In order for the request to be considered valid by a compliant extension apiserver, the following conditions must be met:

1. The connection must be made using a client certificate that is signed by the CA whose certificate is in `--requestheader-client-ca-file`.
2. The connection must be made using a client certificate whose CN is one of those listed in `--requestheader-allowed-names`.

Note: You can set this option to blank as `--requestheader-allowed-names=""`. This will indicate to an extension apiserver that *any* CN is acceptable.

When started with these options, the Kubernetes apiserver will:

1. Use them to authenticate to the extension apiserver.
2. Create a configmap in the `kube-system` namespace called `extension-apiserver-authentication`, in which it will place the CA certificate and the allowed CNs. These in turn can be retrieved by extension apiservers to validate requests.

Note that the same client certificate is used by the Kubernetes apiserver to authenticate against *all* extension apiservers. It does not create a client certificate per extension apiserver, but rather a single one to authenticate as the Kubernetes apiserver. This same one is reused for all extension apiserver requests.

Original Request Username and Group

When the Kubernetes apiserver proxies the request to the extension apiserver, it informs the extension apiserver of the username and group with which the original request successfully authenticated. It provides these in http headers of its proxied request. You must inform the Kubernetes apiserver of the names of the headers to be used.

- the header in which to store the username via `--requestheader-username-headers`
- the header in which to store the group via `--requestheader-group-headers`
- the prefix to append to all extra headers via `--requestheader-extra-headers-prefix`

These header names are also placed in the `extension-apiserver-authentication configmap`, so they can be retrieved and used by extension apiservers.

Extension Apiserver Authenticates the Request

The extension apiserver, upon receiving a proxied request from the Kubernetes apiserver, must validate that the request actually did come from a valid authenticating proxy, which role the Kubernetes apiserver is fulfilling. The extension apiserver validates it via:

1. Retrieve the following from the configmap in `kube-system`, as described above:
 - Client CA certificate
 - List of allowed names (CNs)
 - Header names for username, group and extra info
2. Check that the TLS connection was authenticated using a client certificate which:
 - Was signed by the CA whose certificate matches the retrieved CA certificate.
 - Has a CN in the list of allowed CNs, unless the list is blank, in which case all CNs are allowed.
 - Extract the username and group from the appropriate headers

If the above passes, then the request is a valid proxied request from a legitimate authenticating proxy, in this case the Kubernetes apiserver.

Note that it is the responsibility of the extension apiserver implementation to provide the above. Many do it by default, leveraging the `k8s.io/apiserver/` package. Others may provide options to override it using command-line options.

In order to have permission to retrieve the configmap, an extension apiserver requires the appropriate role. There is a default role named `extension-apiserver-authentication-reader` in the `kube-system` namespace which can be assigned.

Extension Apiserver Authorizes the Request

The extension apiserver now can validate that the user/group retrieved from the headers are authorized to execute the given request. It does so by sending a standard [SubjectAccessReview](#) request to the Kubernetes apiserver.

In order for the extension apiserver to be authorized itself to submit the `SubjectAccessReview` request to the Kubernetes apiserver, it needs the correct permissions. Kubernetes includes a default `ClusterRole` named

`system:auth-delegator` that has the appropriate permissions. It can be granted to the extension apiserver's service account.

Extension Apiserver Executes

If the `SubjectAccessReview` passes, the extension apiserver executes the request.

Enable Kubernetes 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=front-proxy-client
--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>
```

CA Reusage and Conflicts

The Kubernetes apiserver has two client CA options:

- `--client-ca-file`
- `--requestheader-client-ca-file`

Each of these functions independently and can conflict with each other, if not used correctly.

- `--client-ca-file` : When a request arrives to the Kubernetes apiserver, if this option is enabled, the Kubernetes apiserver checks the certificate of the request. If it is signed by one of the CA certificates in the file referenced by `--client-ca-file`, then the request is treated as a legitimate request, and the user is the value of the common name `CN=`, while the group is the organization `O=`. See the [documentation on TLS authentication](#).
- `--requestheader-client-ca-file` : When a request arrives to the Kubernetes apiserver, if this option is enabled, the Kubernetes apiserver checks the certificate of the request. If it is signed by one of the CA certificates in the file reference by `--requestheader-client-ca-file`, then the request is treated as a potentially legitimate request. The Kubernetes apiserver then checks if the common name `CN=` is one of the names in the list provided by `--requestheader-allowed-names`. If the name is allowed, the request is approved; if it is not, the request is not.

If *both* `--client-ca-file` and `--requestheader-client-ca-file` are provided, then the request first checks the `--requestheader-client-ca-file` CA and then the `--client-ca-file`. Normally, different CAs, either root CAs or intermediate CAs, are used for each of these options; regular client requests match against `--client-ca-file`, while aggregation requests match against `--requestheader-client-ca-file`. However, if both use the *same* CA, then client requests that normally would pass via `--client-ca-file` will fail, because the CA will match the CA in `--requestheader-client-ca-file`, but the common name `CN=` will **not** match one of the acceptable common names in `--requestheader-allowed-names`. This can cause your kubelets and other control plane components, as well as end-users, to be unable to authenticate to the Kubernetes apiserver.

For this reason, use different CA certs for the `--client-ca-file` option - to authorize control plane components and end-users - and the `--requestheader-client-ca-file` option - to authorize aggregation apiserver requests.

Warning: Do **not** reuse a CA that is used in a different context unless you understand the risks and the mechanisms to protect the CA's usage.

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 `kube-apiserver` flag:

```
--enable-aggregator-routing=true
```

Register APIService objects

You can dynamically configure what client requests are proxied to extension apiserver. The following is an example registration:

```
apiVersion: apiregistration.k8s.io/v1
kind: APIService
metadata:
  name: <name of the registration object>
spec:
  group: <API group name this extension apiserver hosts>
  version: <API version this extension apiserver hosts>
  groupPriorityMinimum: <priority this APIService for this group, see />
  versionPriority: <prioritizes ordering of this version within a group>
  service:
    namespace: <namespace of the extension apiserver service>
    name: <name of the extension apiserver service>
    caBundle: <pem encoded ca cert that signs the server cert used by the extension apiserver>
```

The name of an APIService object must be a valid [path segment name](#).

Contacting the extension apiserver

Once the Kubernetes apiserver has determined a request should be sent to an extension apiserver, it needs to know how to contact it.

The `service` stanza is a reference to the service for an extension apiserver. The service namespace and name are required. The port is optional and defaults to 443.

Here is an example of an extension apiserver that is configured to be called on port "1234", and to verify the TLS connection against the ServerName `my-service-name.my-service-namespace.svc` using a custom CA bundle.

```
apiVersion: apiregistration.k8s.io/v1
kind: APIService
...
spec:
  ...
  service:
    namespace: my-service-namespace
    name: my-service-name
    port: 1234
    caBundle: "Ci0tLS0tQk...<base64-encoded PEM bundle>...tLS0K"
...
...
```

What's next

- [Set up 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](#).

11.2 - Use Custom Resources

11.2.1 - 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

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

- [KillerCoda](#)
- [Play with Kubernetes](#)

Your Kubernetes server must be at or later than version 1.16. To check the version, enter `kubectl version`. If you are using an older version of Kubernetes that is still supported, switch to the documentation for that version to see advice that is relevant for your cluster.

Create a CustomResourceDefinition

When you create a new CustomResourceDefinition (CRD), the Kubernetes API Server creates a new RESTful resource path for each version you specify. The custom resource created from a CRD object can be either namespaced or cluster-scoped, as specified in the CRD's `spec.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/v1
kind: CustomResourceDefinition
metadata:
  # name must match the spec fields below, and be in the form: <plural>
  name: crontabs.stable.example.com
spec:
  # group name to use for REST API: /apis/<group>/<version>
  group: stable.example.com
  # list of versions supported by this CustomResourceDefinition
  versions:
    - name: v1
      # Each version can be enabled/disabled by Served flag.
      served: true
      # One and only one version must be marked as the storage version
      storage: true
      schema:
        openAPIV3Schema:
          type: object
          properties:
            spec:
              type: object
              properties:
                cronSpec:
                  type: string
                image:
                  type: string
                replicas:
                  type: integer
            # either Namespaced or Cluster
            scope: Namespaced
            names:
              # plural name to be used in the URL: /apis/<group>/<version>/<plural>
              plural: crontabs
              # singular name to be used as an alias on the CLI and for display
              singular: crontab
              # kind is normally the CamelCased singular type. Your resource manager
              kind: CronTab
              # shortNames allow shorter string to match your resource on the CLI
              shortNames:
                - ct
```

and create it:

```
kubectl apply -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.

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 apply -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

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:
    annotations:
      kubectl.kubernetes.io/last-applied-configuration: |
        {"apiVersion":"stable.example.com/v1", "kind":"CronTab", "metadata":{}}
      creationTimestamp: "2021-06-20T07:35:27Z"
    generation: 1
    name: my-new-cron-object
    namespace: default
    resourceVersion: "1326"
    uid: 9aab1d66-628e-41bb-a422-57b8b3b1f5a9
  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 {"stable.example.com": "v1"}
find the requested resource (get crontabs.stable.example.com)
```

If you later recreate the same CustomResourceDefinition, it will start out empty.

Specifying a structural schema

CustomResources store structured data in custom fields (alongside the built-in fields `apiVersion`, `kind` and `metadata`, which the API server validates implicitly). With [OpenAPI v3.0 validation](#) a schema can be specified, which is validated during creation and updates, compare below for details and limits of such a schema.

With `apiextensions.k8s.io/v1` the definition of a structural schema is mandatory for CustomResourceDefinitions. In the beta version of CustomResourceDefinition, the structural schema was optional.

A structural schema is an [OpenAPI v3.0 validation schema](#) which:

1. specifies a non-empty type (via `type` in OpenAPI) for the root, for each specified field of an object node (via `properties` or `additionalProperties` in OpenAPI) and for each item in an array node (via `items` in OpenAPI), with the exception of:
 - o a node with `x-kubernetes-int-or-string: true`

- o a node with `x-kubernetes-preserve-unknown-fields: true`
2. for each field in an object and each item in an array which is specified within any of `allOf`, `anyOf`, `oneOf` or `not`, the schema also specifies the field/item outside of those logical junctors (compare example 1 and 2).
3. does not set `description`, `type`, `default`, `additionalProperties`, `nullable` within an `allOf`, `anyOf`, `oneOf` or `not`, with the exception of the two pattern for `x-kubernetes-int-or-string: true` (see below).
4. if `metadata` is specified, then only restrictions on `metadata.name` and `metadata.generateName` are allowed.

Non-structural example 1:

```
allOf:  
- properties:  
  foo:  
    ...
```

conflicts with rule 2. The following would be correct:

```
properties:  
  foo:  
    ...  
allOf:  
- properties:  
  foo:  
    ...
```

Non-structural example 2:

```
allOf:  
- items:  
  properties:  
    foo:  
      ...
```

conflicts with rule 2. The following would be correct:

```
items:  
  properties:  
    foo:  
      ...  
allOf:  
- items:  
  properties:  
    foo:  
      ...
```

Non-structural example 3:

```

properties:
  foo:
    pattern: "abc"
  metadata:
    type: object
    properties:
      name:
        type: string
        pattern: "^a"
      finalizers:
        type: array
        items:
          type: string
          pattern: "my-finalizer"
anyOf:
- properties:
  bar:
    type: integer
    minimum: 42
  required: ["bar"]
  description: "foo bar object"

```

is not a structural schema because of the following violations:

- the type at the root is missing (rule 1).
- the type of `foo` is missing (rule 1).
- `bar` inside of `anyOf` is not specified outside (rule 2).
- `bar`'s type is within `anyOf` (rule 3).
- the description is set within `anyOf` (rule 3).
- `metadata.finalizers` might not be restricted (rule 4).

In contrast, the following, corresponding schema is structural:

```

type: object
description: "foo bar object"
properties:
  foo:
    type: string
    pattern: "abc"
  bar:
    type: integer
  metadata:
    type: object
    properties:
      name:
        type: string
        pattern: "^a"
anyOf:
- properties:
  bar:
    minimum: 42
  required: ["bar"]

```

Violations of the structural schema rules are reported in the `NonStructural` condition in the CustomResourceDefinition.

Field pruning

CustomResourceDefinitions store validated resource data in the cluster's persistence store, `etcd`. As with native Kubernetes resources such as `ConfigMap`, if you specify a field that the API server does not recognize, the unknown field is *pruned* (removed) before being persisted.

CRDs converted from `apiextensions.k8s.io/v1beta1` to `apiextensions.k8s.io/v1` might lack structural schemas, and `spec.preserveUnknownFields` might be `true`.

For legacy CustomResourceDefinition objects created as `apiextensions.k8s.io/v1beta1` with `spec.preserveUnknownFields` set to `true`, the following is also true:

- Pruning is not enabled.
- You can store arbitrary data.

For compatibility with `apiextensions.k8s.io/v1`, update your custom resource definitions to:

1. Use a structural OpenAPI schema.
2. Set `spec.preserveUnknownFields` to `false`.

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
  someRandomField: 42
```

and create it:

```
kubectl create --validate=false -f my-crontab.yaml -o yaml
```

Your output is similar to:

```
apiVersion: stable.example.com/v1
kind: CronTab
metadata:
  creationTimestamp: 2017-05-31T12:56:35Z
  generation: 1
  name: my-new-cron-object
  namespace: default
  resourceVersion: "285"
  uid: 9423255b-4600-11e7-af6a-28d2447dc82b
spec:
  cronSpec: '* * * * */5'
  image: my-awesome-cron-image
```

Notice that the field `someRandomField` was pruned.

This example turned off client-side validation to demonstrate the API server's behavior, by adding the `--validate=false` command line option. Because the [OpenAPI validation schemas are also published](#) to clients, `kubectl` also checks for unknown fields and rejects those objects well before they would be sent to the API server.

Controlling pruning

By default, all unspecified fields for a custom resource, across all versions, are pruned. It is possible though to opt-out of that for specific sub-trees of fields by adding `x-kubernetes-preserve-unknown-fields: true` in the [structural OpenAPI v3 validation schema](#).

For example:

```
type: object
properties:
  json:
    x-kubernetes-preserve-unknown-fields: true
```

The field `json` can store any JSON value, without anything being pruned.

You can also partially specify the permitted JSON; for example:

```
type: object
properties:
  json:
    x-kubernetes-preserve-unknown-fields: true
    type: object
    description: this is arbitrary JSON
```

With this, only `object` type values are allowed.

Pruning is enabled again for each specified property (or `additionalProperties`):

```
type: object
properties:
  json:
    x-kubernetes-preserve-unknown-fields: true
    type: object
    properties:
      spec:
        type: object
        properties:
          foo:
            type: string
          bar:
            type: string
```

With this, the value:

```
json:
  spec:
    foo: abc
    bar: def
    something: x
  status:
    something: x
```

is pruned to:

```
json:
  spec:
    foo: abc
    bar: def
  status:
    something: x
```

This means that the `something` field in the specified `spec` object is pruned, but everything outside is not.

IntOrString

Nodes in a schema with `x-kubernetes-int-or-string: true` are excluded from rule 1, such that the following is structural:

```
type: object
properties:
  foo:
    x-kubernetes-int-or-string: true
```

Also those nodes are partially excluded from rule 3 in the sense that the following two patterns are allowed (exactly those, without variations in order to additional fields):

```
x-kubernetes-int-or-string: true
anyOf:
  - type: integer
  - type: string
...
...
```

and

```
x-kubernetes-int-or-string: true
allOf:
  - anyOf:
    - type: integer
    - type: string
  - ... # zero or more
...
...
```

With one of those specification, both an integer and a string validate.

In [Validation Schema Publishing](#), `x-kubernetes-int-or-string: true` is unfolded to one of the two patterns shown above.

RawExtension

RawExtensions (as in [`runtime.RawExtension`](#)) holds complete Kubernetes objects, i.e. with `apiVersion` and `kind` fields.

It is possible to specify those embedded objects (both completely without constraints or partially specified) by setting `x-kubernetes-embedded-resource: true`. For example:

```
type: object
properties:
  foo:
    x-kubernetes-embedded-resource: true
    x-kubernetes-preserve-unknown-fields: true
```

Here, the field `foo` holds a complete object, e.g.:

```
foo:
  apiVersion: v1
  kind: Pod
  spec:
    ...
```

Because `x-kubernetes-preserve-unknown-fields: true` is specified alongside, nothing is pruned. The use of `x-kubernetes-preserve-unknown-fields: true` is optional though.

With `x-kubernetes-embedded-resource: true`, the `apiVersion`, `kind` and `metadata` are implicitly specified and validated.

Serving multiple versions of a CRD

See [Custom resource definition versioning](#) for more information about serving multiple versions of your CustomResourceDefinition and migrating your objects from one version to another.

Advanced topics

Finalizers

Finalizers allow controllers to implement asynchronous pre-delete hooks. Custom objects support finalizers similar to built-in objects.

You can add a finalizer to a custom object like this:

```
apiVersion: "stable.example.com/v1"
kind: CronTab
metadata:
  finalizers:
    - stable.example.com/finalizer
```

Identifiers of custom finalizers consist of a domain name, a forward slash and the name of the finalizer. Any controller can add a finalizer to any object's list of finalizers.

The first delete request on an object with finalizers sets a value for the `metadata.deletionTimestamp` field but does not delete it. Once this value is set, entries in the `finalizers` list can only be removed. While any finalizers remain it is also impossible to force the deletion of an object.

When the `metadata.deletionTimestamp` field is set, controllers watching the object execute any finalizers they handle and remove the finalizer from the list after they are done. It is the responsibility of each controller to remove its finalizer from the list.

The value of `metadata.deletionGracePeriodSeconds` controls the interval between polling updates.

Once the list of finalizers is empty, meaning all finalizers have been executed, the resource is deleted by Kubernetes.

Validation

Custom resources are validated via [OpenAPI v3 schemas](#), by x-kubernetes-validations when the [Validation Rules feature](#) is enabled, and you can add additional validation using [admission webhooks](#).

Additionally, the following restrictions are applied to the schema:

- These fields cannot be set:
 - `definitions`,
 - `dependencies`,
 - `deprecated`,
 - `discriminator`,
 - `id`,
 - `patternProperties`,
 - `readOnly`,
 - `writeOnly`,
 - `xml`,
 - `$ref`.
- The field `uniqueItems` cannot be set to `true`.
- The field `additionalProperties` cannot be set to `false`.
- The field `additionalProperties` is mutually exclusive with `properties`.

The `x-kubernetes-validations` extension can be used to validate custom resources using [Common Expression Language \(CEL\)](#) expressions when the [Validation rules](#) feature is enabled and the CustomResourceDefinition schema is a [structural schema](#).

Refer to the [structural schemas](#) section for other restrictions and CustomResourceDefinition features.

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/v1
kind: CustomResourceDefinition
metadata:
  name: crontabs.stable.example.com
spec:
  group: stable.example.com
  versions:
    - name: v1
      served: true
      storage: true
      schema:
        # openAPIV3Schema is the schema for validating custom objects.
        openAPIV3Schema:
          type: object
          properties:
            spec:
              type: object
              properties:
                cronSpec:
                  type: string
                  pattern: '^(\d+|*)(/(\d+)?(\s+(\d+|*)(/(\d+)?){4})$'
            image:
              type: string
            replicas:
              type: integer
              minimum: 1
              maximum: 10
  scope: Namespaced
  names:
    plural: crontabs
    singular: crontab
    kind: CronTab
    shortNames:
      - ct

```

and create it:

```
kubectl apply -f resourcedefinition.yaml
```

A request to create a custom object of kind CronTab is 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 attempt to create it:

```
kubectl apply -f my-crontab.yaml
```

then you get an error:

```
The CronTab "my-new-cron-object" is invalid: []: Invalid value: map[st
validation failure list:
spec.cronSpec in body should match '^(\d+|\*)(/\d+)?(\s+(\d+|\*)(/\d+
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 apply -f my-crontab.yaml
crontab "my-new-cron-object" created
```

Validation ratcheting

FEATURE STATE: [Kubernetes v1.28 \[alpha\]](#)

You need to enable the `CRDValidationRatcheting` [feature gate](#) to use this behavior, which then applies to all CustomResourceDefinitions in your cluster.

Provided you enabled the feature gate, Kubernetes implements *validation ratcheting* for CustomResourceDefinitions. The API server is willing to accept updates to resources that are not valid after the update, provided that each part of the resource that failed to validate was not changed by the update operation. In other words, any invalid part of the resource that remains invalid must have already been wrong. You cannot use this mechanism to update a valid resource so that it becomes invalid.

This feature allows authors of CRDs to confidently add new validations to the OpenAPIV3 schema under certain conditions. Users can update to the new schema safely without bumping the version of the object or breaking workflows.

While most validations placed in the OpenAPIV3 schema of a CRD support ratcheting, there are a few exceptions. The following OpenAPIV3 schema validations are not supported by ratcheting under the implementation in Kubernetes 1.28 and if violated will continue to throw an error as normally:

- Quantors
 - allOf
 - oneOf
 - anyOf

- not
- any validations in a descendent of one of these fields
- `x-kubernetes-validations` For Kubernetes 1.28, CRD validation rules] (#validation-rules) are ignored by ratcheting. This may change in later Kubernetes releases.
- `x-kubernetes-list-type` Errors arising from changing the list type of a subschema will not be ratcheted. For example adding `set` onto a list with duplicates will always result in an error.
- `x-kubernetes-map-keys` Errors arising from changing the map keys of a list schema will not be ratcheted.
- `required` Errors arising from changing the list of required fields will not be ratcheted.
- `properties` Adding/removing/modifying the names of properties is not ratcheted, but changes to validations in each properties' schemas and subschemas may be ratcheted if the name of the property stays the same.
- `additionalProperties` To remove a previously specified `additionalProperties` validation will not be ratcheted.

Validation rules

FEATURE STATE: [Kubernetes v1.25 \[beta\]](#)

Validation rules are in beta since 1.25 and the `CustomResourceValidationExpressions` [feature gate](#) is enabled by default to validate custom resource based on *validation rules*. You can disable this feature by explicitly setting the `CustomResourceValidationExpressions` feature gate to `false`, for the [kube-apiserver](#) component. This feature is only available if the schema is a [structural schema](#).

Validation rules use the [Common Expression Language \(CEL\)](#) to validate custom resource values. Validation rules are included in `CustomResourceDefinition` schemas using the `x-kubernetes-validations` extension.

The Rule is scoped to the location of the `x-kubernetes-validations` extension in the schema. And `self` variable in the CEL expression is bound to the scoped value.

All validation rules are scoped to the current object: no cross-object or stateful validation rules are supported.

For example:

```

...
openAPIV3Schema:
  type: object
  properties:
    spec:
      type: object
      x-kubernetes-validations:
        - rule: "self.minReplicas <= self.replicas"
          message: "replicas should be greater than or equal to minReplicas"
        - rule: "self.replicas <= self.maxReplicas"
          message: "replicas should be smaller than or equal to maxReplicas"
      properties:
        ...
        minReplicas:
          type: integer
        replicas:
          type: integer
        maxReplicas:
          type: integer
      required:
        - minReplicas
        - replicas
        - maxReplicas

```

will reject a request to create this custom resource:

```

apiVersion: "stable.example.com/v1"
kind: CronTab
metadata:
  name: my-new-cron-object
spec:
  minReplicas: 0
  replicas: 20
  maxReplicas: 10

```

with the response:

```

The CronTab "my-new-cron-object" is invalid:
* spec: Invalid value: map[string]interface {}{"maxReplicas":10, "minReplicas":0, "replicas":20}

```

`x-kubernetes-validations` could have multiple rules. The `rule` under `x-kubernetes-validations` represents the expression which will be evaluated by CEL. The `message` represents the message displayed when validation fails. If `message` is unset, the above response would be:

```

The CronTab "my-new-cron-object" is invalid:
* spec: Invalid value: map[string]interface {}{"maxReplicas":10, "minReplicas":0, "replicas":20}

```

Note: You can quickly test CEL expressions in [CEL Playground](#).

Validation rules are compiled when CRDs are created/updated. The request of CRDs create/update will fail if compilation of validation rules fail. Compilation process includes type checking as well.

The compilation failure:

- `no_matching_overload` : this function has no overload for the types of the arguments.

For example, a rule like `self == true` against a field of integer type will get error:

```
Invalid value: apiextensions.ValidationRule{Rule:"self == true", Mes...
```

- `no_such_field` : does not contain the desired field.

For example, a rule like `self.nonExistingField > 0` against a non-existing field will return the following error:

```
Invalid value: apiextensions.ValidationRule{Rule:"self.nonExistingF...
```

- `invalid_argument` : invalid argument to macros.

For example, a rule like `has(self)` will return error:

```
Invalid value: apiextensions.ValidationRule{Rule:"has(self)", Mes...
```

Validation Rules Examples:

Rule	Purpose
<code>self.minReplicas <= self.replicas && self.replicas <= self.maxReplicas</code>	Validate that the three fields defining replicas are ordered appropriately
'Available' in <code>self.stateCounts</code>	Validate that an entry with the 'Available' key exists in a map
<code>(size(self.list1) == 0) != (size(self.list2) == 0)</code>	Validate that one of two lists is non-empty, but not both
<code>!('MY_KEY' in self.map1) self['MY_KEY'].matches('^[a-zA-Z]*\$')</code>	Validate the value of a map for a specific key, if it is in the map
<code>self.envars.filter(e, e.name == 'MY_ENV').all(e, e.value.matches('^[a-zA-Z]*\$'))</code>	Validate the 'value' field of a listMap entry where key field 'name' is 'MY_ENV'
<code>has(self.expired) && self.created + self.ttl < self.expired</code>	Validate that 'expired' date is after a 'create' date plus a 'ttl' duration
<code>self.health.startsWith('ok')</code>	Validate a 'health' string field has the prefix 'ok'
<code>self.widgets.exists(w, w.key == 'x' && w.foo < 10)</code>	Validate that the 'foo' property of a listMap item with a key 'x' is less than 10
<code>type(self) == string ? self == '100%' : self == 1000</code>	Validate an int-or-string field for both the int and string cases

Rule	Purpose
<code>self.metadata.name.startsWith(self.prefix)</code>	Validate that an object's name has the prefix of another field value
<code>self.set1.all(e, !(e in self.set2))</code>	Validate that two listSets are disjoint
<code>size(self.names) == size(self.details) && self.names.all(n, n in self.details)</code>	Validate the 'details' map is keyed by the items in the 'names' listSet
<code>size(self.clusters.filter(c, c.name == self.primary)) == 1</code>	Validate that the 'primary' property has one and only one occurrence in the 'clusters' listMap

Xref: [Supported evaluation on CEL](#)

- If the Rule is scoped to the root of a resource, it may make field selection into any fields declared in the OpenAPIv3 schema of the CRD as well as `apiVersion`, `kind`, `metadata.name` and `metadata.generateName`. This includes selection of fields in both the `spec` and `status` in the same expression:

```
...
openAPIV3Schema:
  type: object
  x-kubernetes-validations:
    - rule: "self.status.availableReplicas >= self.spec.minReplicas"
  properties:
    spec:
      type: object
      properties:
        minReplicas:
          type: integer
        ...
    status:
      type: object
      properties:
        availableReplicas:
          type: integer
```

- If the Rule is scoped to an object with properties, the accessible properties of the object are field selectable via `self.field` and field presence can be checked via `has(self.field)`. Null valued fields are treated as absent fields in CEL expressions.

```
...
openAPIV3Schema:
  type: object
  properties:
    spec:
      type: object
      x-kubernetes-validations:
        - rule: "has(self.foo)"
    properties:
      ...
    foo:
      type: integer
```

- If the Rule is scoped to an object with additionalProperties (i.e. a map) the value of the map are accessible via `self[mapKey]`, map containment can be checked via `mapKey` in `self` and all entries of the map are accessible via CEL macros and functions such as `self.all(...)`.

```

...
openAPIV3Schema:
  type: object
  properties:
    spec:
      type: object
      x-kubernetes-validations:
        - rule: "self['xyz'].foo > 0"
    additionalProperties:
      ...
      type: object
      properties:
        foo:
          type: integer

```

- If the Rule is scoped to an array, the elements of the array are accessible via `self[i]` and also by macros and functions.

```

...
openAPIV3Schema:
  type: object
  properties:
    ...
    foo:
      type: array
      x-kubernetes-validations:
        - rule: "size(self) == 1"
      items:
        type: string

```

- If the Rule is scoped to a scalar, `self` is bound to the scalar value.

```

...
openAPIV3Schema:
  type: object
  properties:
    spec:
      type: object
      properties:
        ...
        foo:
          type: integer
          x-kubernetes-validations:
            - rule: "self > 0"

```

Examples:

type of the field rule scoped to	Rule example
root object	<code>self.status.actual <= self.spec.maxDesired</code>
map of objects	<code>self.components['Widget'].priority < 10</code>
list of integers	<code>self.values.all(value, value >= 0 && value < 100)</code>
string	<code>self.startsWith('kube')</code>

The `apiVersion`, `kind`, `metadata.name` and `metadata.generateName` are always accessible from the root of the object and from any `x-kubernetes-embedded-resource` annotated objects. No other metadata properties are accessible.

Unknown data preserved in custom resources via `x-kubernetes-preserve-unknown-fields` is not accessible in CEL expressions. This includes:

- Unknown field values that are preserved by object schemas with `x-kubernetes-preserve-unknown-fields`.
- Object properties where the property schema is of an "unknown type". An "unknown type" is recursively defined as:
 - A schema with no type and `x-kubernetes-preserve-unknown-fields` set to true
 - An array where the items schema is of an "unknown type"
 - An object where the `additionalProperties` schema is of an "unknown type"

Only property names of the form `[a-zA-Z_-.][a-zA-Z0-9_-.]*` are accessible. Accessible property names are escaped according to the following rules when accessed in the expression:

escape sequence	property name equivalent
<code>_underscores_</code>	<code>_</code>
<code>_dot_</code>	<code>.</code>
<code>_dash_</code>	<code>-</code>
<code>_slash_</code>	<code>/</code>
<code>_{{keyword}}_</code>	CEL RESERVED keyword

Note: CEL RESERVED keyword needs to match the exact property name to be escaped (e.g. `int` in the word `sprint` would not be escaped).

Examples on escaping:

property name	rule with escaped property name
<code>namespace</code>	<code>self._namespace_ > 0</code>
<code>x-prop</code>	<code>self.x_dash_prop > 0</code>
<code>redact_d</code>	<code>self.redact_underscores_d > 0</code>
<code>string</code>	<code>self.startsWith('kube')</code>

Equality on arrays with `x-kubernetes-list-type` of `set` or `map` ignores element order, i.e., `[1, 2] == [2, 1]`. Concatenation on arrays with `x-kubernetes-list-type` use the semantics of the list type:

- `set` : $x + y$ performs a union where the array positions of all elements in x are preserved and non-intersecting elements in y are appended, retaining their partial order.
- `map` : $x + y$ performs a merge where the array positions of all keys in x are preserved but the values are overwritten by values in y when the key sets of x and y intersect. Elements in y with non-intersecting keys are appended, retaining their partial order.

Here is the declarations type mapping between OpenAPIv3 and CEL type:

OpenAPIv3 type	CEL type
'object' with Properties	object / "message type"
'object' with AdditionalProperties	map
'object' with x-kubernetes-embedded-type	object / "message type", 'apiVersion', 'kind', 'metadata.name' and 'metadata.generateName' are implicitly included in schema
'object' with x-kubernetes-preserve-unknown-fields	object / "message type", unknown fields are NOT accessible in CEL expression
x-kubernetes-int-or-string	dynamic object that is either an int or a string, <code>type(value)</code> can be used to check the type
'array'	list
'array' with x-kubernetes-list-type=map	list with map based Equality & unique key guarantees
'array' with x-kubernetes-list-type=set	list with set based Equality & unique entry guarantees
'boolean'	boolean
'number' (all formats)	double
'integer' (all formats)	int (64)
'null'	null_type
'string'	string
'string' with format=byte (base64 encoded)	bytes
'string' with format=date	timestamp (google.protobuf.Timestamp)
'string' with format=datetime	timestamp (google.protobuf.Timestamp)
'string' with format=duration	duration (google.protobuf.Duration)

xref: [CEL types](#), [OpenAPI types](#), [Kubernetes Structural Schemas](#).

The messageExpression field

Similar to the `message` field, which defines the string reported for a validation rule failure, `messageExpression` allows you to use a CEL expression to construct the message string. This allows you to insert more descriptive information into the validation failure message.

`messageExpression` must evaluate a string and may use the same variables that are available to the `rule` field. For example:

```
x-kubernetes-validations:  
- rule: "self.x <= self.maxLimit"  
  messageExpression: '"x exceeded max limit of " + string(self.maxLimit)"'
```

Keep in mind that CEL string concatenation (+ operator) does not auto-cast to string. If you have a non-string scalar, use the `string(<value>)` function to cast the scalar to a string like shown in the above example.

`messageExpression` must evaluate to a string, and this is checked while the CRD is being written. Note that it is possible to set `message` and `messageExpression` on the same rule, and if both are present, `messageExpression` will be used. However, if `messageExpression` evaluates to an error, the string defined in `message` will be used instead, and the `messageExpression` error will be logged. This fallback will also occur if the CEL expression defined in `messageExpression` generates an empty string, or a string containing line breaks.

If one of the above conditions are met and no `message` has been set, then the default validation failure message will be used instead.

`messageExpression` is a CEL expression, so the restrictions listed in [Resource use by validation functions](#) apply. If evaluation halts due to resource constraints during `messageExpression` execution, then no further validation rules will be executed.

Setting `messageExpression` is optional.

The `message` field

If you want to set a static message, you can supply `message` rather than `messageExpression`. The value of `message` is used as an opaque error string if validation fails.

Setting `message` is optional.

The `reason` field

You can add a machine-readable validation failure reason within a `validation`, to be returned whenever a request fails this validation rule.

For example:

```
x-kubernetes-validations:  
- rule: "self.x <= self.maxLimit"  
  reason: "FieldValueInvalid"
```

The HTTP status code returned to the caller will match the reason of the first failed validation rule. The currently supported reasons are: "FieldValueInvalid", "FieldValueForbidden", "FieldValueRequired", "FieldValueDuplicate". If not set or unknown reasons, default to use "FieldValueInvalid".

Setting `reason` is optional.

The `fieldPath` field

You can specify the field path returned when the validation fails.

For example:

```
x-kubernetes-validations:
- rule: "self.foo.test.x <= self.maxLimit"
  fieldPath: ".foo.test.x"
```

In the example above, the validation checks the value of field `x` should be less than the value of `maxLimit`. If no `fieldPath` specified, when validation fails, the `fieldPath` would be default to wherever `self` scoped. With `fieldPath` specified, the returned error will have `fieldPath` properly refer to the location of field `x`.

The `fieldPath` value must be a relative JSON path that is scoped to the location of this `x-kubernetes-validations` extension in the schema. Additionally, it should refer to an existing field within the schema. For example when validation checks if a specific attribute `foo` under a map `testMap`, you could set `fieldPath` to `".testMap.foo"` or `.testMap['foo']`. If the validation requires checking for unique attributes in two lists, the `fieldPath` can be set to either of the lists. For example, it can be set to `.testList1` or `.testList2`. It supports child operation to refer to an existing field currently. Refer to [JSONPath support in Kubernetes](#) for more info. The `fieldPath` field does not support indexing arrays numerically.

Setting `fieldPath` is optional.

Validation functions

Functions available include:

- CEL standard functions, defined in the [list of standard definitions](#)
- CEL standard [macros](#)
- CEL [extended string function library](#)
- Kubernetes [CEL extension library](#)

Transition rules

A rule that contains an expression referencing the identifier `oldSelf` is implicitly considered a *transition rule*. Transition rules allow schema authors to prevent certain transitions between two otherwise valid states. For example:

```
type: string
enum: ["low", "medium", "high"]
x-kubernetes-validations:
- rule: "!(self == 'high' && oldSelf == 'low') && !(self == 'low' && o
  message: cannot transition directly between 'low' and 'high'
```

Unlike other rules, transition rules apply only to operations meeting the following criteria:

- The operation updates an existing object. Transition rules never apply to create operations.
- Both an old and a new value exist. It remains possible to check if a value has been added or removed by placing a transition rule on the parent node. Transition rules are never applied to custom resource creation. When placed on an optional field, a transition rule will not apply to update operations that set or unset the field.

- The path to the schema node being validated by a transition rule must resolve to a node that is comparable between the old object and the new object. For example, list items and their descendants (`spec.foo[10].bar`) can't necessarily be correlated between an existing object and a later update to the same object.

Errors will be generated on CRD writes if a schema node contains a transition rule that can never be applied, e.g. "`path: update rule rule` cannot be set on schema because the schema or its parent schema is not mergeable".

Transition rules are only allowed on *correlatable portions* of a schema. A portion of the schema is correlatable if all `array` parent schemas are of type `x-kubernetes-list-type=map`; any `set` or `atomic` array parent schemas make it impossible to unambiguously correlate a `self` with `oldSelf`.

Here are some examples for transition rules:

Use Case	Rule
Immutability	<code>self.foo == oldSelf.foo</code>
Prevent modification/removal once assigned	<code>oldSelf != 'bar' self == 'bar' or !has(oldSelf.field) has(self.field)</code>
Append-only set	<code>self.all(element, element in oldSelf)</code>
If previous value was X, new value can only be A or B, not Y or Z	<code>oldSelf != 'X' self in ['A', 'B']</code>
Monotonic (non-decreasing) counters	<code>self >= oldSelf</code>

Resource use by validation functions

When you create or update a CustomResourceDefinition that uses validation rules, the API server checks the likely impact of running those validation rules. If a rule is estimated to be prohibitively expensive to execute, the API server rejects the create or update operation, and returns an error message. A similar system is used at runtime that observes the actions the interpreter takes. If the interpreter executes too many instructions, execution of the rule will be halted, and an error will result. Each CustomResourceDefinition is also allowed a certain amount of resources to finish executing all of its validation rules. If the sum total of its rules are estimated at creation time to go over that limit, then a validation error will also occur.

You are unlikely to encounter issues with the resource budget for validation if you only specify rules that always take the same amount of time regardless of how large their input is. For example, a rule that asserts that `self.foo == 1` does not by itself have any risk of rejection on validation resource budget groups. But if `foo` is a string and you define a validation rule `self.foo.contains("someString")`, that rule takes longer to execute depending on how long `foo` is. Another example would be if `foo` were an array, and you specified a validation rule `self.foo.all(x, x > 5)`. The cost system always assumes the worst-case scenario if a limit on the length of `foo` is not given, and this will happen for anything that can be iterated over (lists, maps, etc.).

Because of this, it is considered best practice to put a limit via `maxItems`, `maxProperties`, and `maxLength` for anything that will be processed in a validation rule in order to prevent validation errors during cost estimation. For example, given this schema with one rule:

```
openAPIV3Schema:
  type: object
  properties:
    foo:
      type: array
      items:
        type: string
      x-kubernetes-validations:
        - rule: "self.all(x, x.contains('a string'))"
```

then the API server rejects this rule on validation budget grounds with error:

```
spec.validation.openAPIV3Schema.properties[spec].properties[foo].x-kube
CEL rule exceeded budget by more than 100x (try simplifying the rule, o
maxLength where arrays, maps, and strings are used)
```

The rejection happens because `self.all` implies calling `contains()` on every string in `foo`, which in turn will check the given string to see if it contains `'a string'`. Without limits, this is a very expensive rule.

If you do not specify any validation limit, the estimated cost of this rule will exceed the per-rule cost limit. But if you add limits in the appropriate places, the rule will be allowed:

```
openAPIV3Schema:
  type: object
  properties:
    foo:
      type: array
      maxItems: 25
      items:
        type: string
        maxLength: 10
      x-kubernetes-validations:
        - rule: "self.all(x, x.contains('a string'))"
```

The cost estimation system takes into account how many times the rule will be executed in addition to the estimated cost of the rule itself. For instance, the following rule will have the same estimated cost as the previous example (despite the rule now being defined on the individual array items):

```
openAPIV3Schema:
  type: object
  properties:
    foo:
      type: array
      maxItems: 25
      items:
        type: string
      x-kubernetes-validations:
        - rule: "self.contains('a string'))"
      maxLength: 10
```

If a list inside of a list has a validation rule that uses `self.all`, that is significantly more expensive than a non-nested list with the same rule. A rule that would have been allowed on a non-nested list might need lower

limits set on both nested lists in order to be allowed. For example, even without having limits set, the following rule is allowed:

```
openAPIV3Schema:  
  type: object  
  properties:  
    foo:  
      type: array  
      items:  
        type: integer  
    x-kubernetes-validations:  
      - rule: "self.all(x, x == 5)"
```

But the same rule on the following schema (with a nested array added) produces a validation error:

```
openAPIV3Schema:  
  type: object  
  properties:  
    foo:  
      type: array  
      items:  
        type: array  
        items:  
          type: integer  
    x-kubernetes-validations:  
      - rule: "self.all(x, x == 5)"
```

This is because each item of `foo` is itself an array, and each subarray in turn calls `self.all`. Avoid nested lists and maps if possible where validation rules are used.

Defaulting

Note: To use defaulting, your CustomResourceDefinition must use API version `apiextensions.k8s.io/v1`.

Defaulting allows to specify default values in the [OpenAPI v3 validation schema](#):

```

apiVersion: apiextensions.k8s.io/v1
kind: CustomResourceDefinition
metadata:
  name: crontabs.stable.example.com
spec:
  group: stable.example.com
  versions:
    - name: v1
      served: true
      storage: true
      schema:
        # openAPIV3Schema is the schema for validating custom objects.
        openAPIV3Schema:
          type: object
          properties:
            spec:
              type: object
              properties:
                cronSpec:
                  type: string
                  pattern: '^(\d+|*)(/d+)?(\s+(\d+|*)(/d+)?){4}$'
                  default: "5 0 * * *"
                image:
                  type: string
            replicas:
              type: integer
              minimum: 1
              maximum: 10
              default: 1
  scope: Namespaced
  names:
    plural: crontabs
    singular: crontab
    kind: CronTab
    shortNames:
      - ct

```

With this both `cronSpec` and `replicas` are defaulted:

```

apiVersion: "stable.example.com/v1"
kind: CronTab
metadata:
  name: my-new-cron-object
spec:
  image: my-awesome-cron-image

```

leads to

```

apiVersion: "stable.example.com/v1"
kind: CronTab
metadata:
  name: my-new-cron-object
spec:
  cronSpec: "5 0 * * *"
  image: my-awesome-cron-image
  replicas: 1

```

Defaulting happens on the object

- in the request to the API server using the request version defaults,

- when reading from etcd using the storage version defaults,
- after mutating admission plugins with non-empty patches using the admission webhook object version defaults.

Defaults applied when reading data from etcd are not automatically written back to etcd. An update request via the API is required to persist those defaults back into etcd.

Default values must be pruned (with the exception of defaults for `metadata` fields) and must validate against a provided schema.

Default values for `metadata` fields of `x-kubernetes-embedded-resources`: `true` nodes (or parts of a default value covering `metadata`) are not pruned during CustomResourceDefinition creation, but through the pruning step during handling of requests.

Defaulting and Nullable

Null values for fields that either don't specify the nullable flag, or give it a `false` value, will be pruned before defaulting happens. If a default is present, it will be applied. When nullable is `true`, null values will be conserved and won't be defaulted.

For example, given the OpenAPI schema below:

```
type: object
properties:
  spec:
    type: object
    properties:
      foo:
        type: string
        nullable: false
        default: "default"
      bar:
        type: string
        nullable: true
      baz:
        type: string
```

creating an object with null values for `foo` and `bar` and `baz`

```
spec:
  foo: null
  bar: null
  baz: null
```

leads to

```
spec:
  foo: "default"
  bar: null
```

with `foo` pruned and defaulted because the field is non-nullable, `bar` maintaining the null value due to `nullable: true`, and `baz` pruned because the field is non-nullable and has no default.

Publish Validation Schema in OpenAPI

CustomResourceDefinition [OpenAPI v3 validation schemas](#) which are [structural](#) and [enable pruning](#) are published as [OpenAPI v3](#) and [OpenAPI v2](#) from Kubernetes API server. It is recommended to use the OpenAPI v3 document as it is a lossless representation of the CustomResourceDefinition OpenAPI v3 validation schema while OpenAPI v2 represents a lossy conversion.

The [kubectl](#) command-line tool consumes the published schema to perform client-side validation (`kubectl create` and `kubectl apply`), schema explanation (`kubectl explain`) on custom resources. The published schema can be consumed for other purposes as well, like client generation or documentation.

Compatibility with OpenAPI V2

For compatibility with OpenAPI V2, the OpenAPI v3 validation schema performs a lossy conversion to the OpenAPI v2 schema. The schema show up in `definitions` and `paths` fields in the [OpenAPI v2 spec](#).

The following modifications are applied during the conversion to keep backwards compatibility with kubectl in previous 1.13 version. These modifications prevent kubectl from being over-strict and rejecting valid OpenAPI schemas that it doesn't understand. The conversion won't modify the validation schema defined in CRD, and therefore won't affect [validation](#) in the API server.

1. The following fields are removed as they aren't supported by OpenAPI v2.
 - o The fields `allOf`, `anyOf`, `oneOf` and `not` are removed
2. If `nullable: true` is set, we drop `type`, `nullable`, `items` and `properties` because OpenAPI v2 is not able to express nullable. To avoid kubectl to reject good objects, this is necessary.

Additional printer columns

The kubectl tool relies on server-side output formatting. Your cluster's API server decides which columns are shown by the `kubectl get` command. You can customize these columns for a CustomResourceDefinition. The following example adds the `Spec`, `Replicas`, and `Age` columns.

Save the CustomResourceDefinition to `resourcedefinition.yaml`:

```

apiVersion: apiextensions.k8s.io/v1
kind: CustomResourceDefinition
metadata:
  name: crontabs.stable.example.com
spec:
  group: stable.example.com
  scope: Namespaced
  names:
    plural: crontabs
    singular: crontab
    kind: CronTab
    shortNames:
      - ct
  versions:
    - name: v1
      served: true
      storage: true
      schema:
        openAPIV3Schema:
          type: object
          properties:
            spec:
              type: object
              properties:
                cronSpec:
                  type: string
                image:
                  type: string
                replicas:
                  type: integer
            additionalPrinterColumns:
              - name: Spec
                type: string
                description: The cron spec defining the interval a CronJob is run
                jsonPath: .spec.cronSpec
              - name: Replicas
                type: integer
                description: The number of jobs launched by the CronJob
                jsonPath: .spec.replicas
              - name: Age
                type: date
                jsonPath: .metadata.creationTimestamp

```

◀ ▶

Create the CustomResourceDefinition:

```
kubectl apply -f resourcedefinition.yaml
```

Create an instance using the `my-crontab.yaml` from the previous section.

Invoke the server-side printing:

```
kubectl get crontab my-new-cron-object
```

Notice the `NAME`, `SPEC`, `REPLICAS`, and `AGE` columns in the output:

NAME	SPEC	REPLICAS	AGE
my-new-cron-object	* * * * *	1	7s

Note: The `NAME` column is implicit and does not need to be defined in the CustomResourceDefinition.

Priority

Each column includes a `priority` field. Currently, the priority differentiates between columns shown in standard view or wide view (using the `-o wide` flag).

- Columns with priority `0` are shown in standard view.
- Columns with priority greater than `0` are shown only in wide view.

Type

A column's `type` field can be any of the following (compare [OpenAPI v3 data types](#)):

- `integer` – non-floating-point numbers
- `number` – floating point numbers
- `string` – strings
- `boolean` – `true` OR `false`
- `date` – rendered differentially as time since this timestamp.

If the value inside a CustomResource does not match the type specified for the column, the value is omitted. Use CustomResource validation to ensure that the value types are correct.

Format

A column's `format` field can be any of the following:

- `int32`
- `int64`
- `float`
- `double`
- `byte`
- `date`
- `date-time`
- `password`

The column's `format` controls the style used when `kubectl` prints the value.

Subresources

Custom resources support `/status` and `/scale` subresources.

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.
- The .metadata.generation value is incremented for all changes, except for changes to .metadata or .status .
- Only the following constructs are allowed at the root of the CRD OpenAPI validation schema:
 - description
 - example
 - exclusiveMaximum
 - exclusiveMinimum
 - externalDocs
 - format
 - items
 - maximum
 - maxItems
 - maxLength
 - minimum
 - minItems
 - minLength
 - multipleOf
 - pattern
 - properties
 - required
 - title
 - type
 - uniqueItems

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 fields 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 .

- o It is an optional value.
- o It must be set to work with HPA and VPA.
- o Only JSONPaths under `.status` or `.spec` and with the dot notation are allowed.
- o 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.
- o The field pointed by this JSON path must be a string field (not a complex selector struct) which contains a serialized label selector in string form.

In the following example, both status and scale subresources are enabled.

Save the CustomResourceDefinition to `resourcedefinition.yaml`:

```

apiVersion: apiextensions.k8s.io/v1
kind: CustomResourceDefinition
metadata:
  name: crontabs.stable.example.com
spec:
  group: stable.example.com
  versions:
    - name: v1
      served: true
      storage: true
      schema:
        openAPIV3Schema:
          type: object
          properties:
            spec:
              type: object
              properties:
                cronSpec:
                  type: string
                image:
                  type: string
                replicas:
                  type: integer
            status:
              type: object
              properties:
                replicas:
                  type: integer
                labelSelector:
                  type: string
  # 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's spec for the replicas field.
      specReplicasPath: .spec.replicas
      # statusReplicasPath defines the JSONPath inside of a custom resource's status for the replicas field.
      statusReplicasPath: .status.replicas
      # labelSelectorPath defines the JSONPath inside of a custom resource's status for the label selector field.
      labelSelectorPath: .status.labelSelector
  scope: Namespaced
  names:
    plural: crontabs
    singular: crontab
    kind: CronTab
    shortNames:
      - ct

```

And create it:

```
kubectl apply -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 apply -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
```

You can use a [PodDisruptionBudget](#) to protect custom resources that have the scale subresource enabled.

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.

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/v1
kind: CustomResourceDefinition
metadata:
  name: crontabs.stable.example.com
spec:
  group: stable.example.com
  versions:
    - name: v1
      served: true
      storage: true
      schema:
        openAPIV3Schema:
          type: object
          properties:
            spec:
              type: object
              properties:
                cronSpec:
                  type: string
                image:
                  type: string
                replicas:
                  type: integer
  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 apply -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 apply -f my-crontab.yaml
```

You can specify the category when 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

- Read about [custom resources](#).
- See [CustomResourceDefinition](#).
- Serve [multiple versions](#) of a CustomResourceDefinition.

11.2.2 - Versions in CustomResourceDefinitions

This page explains how to add versioning information to [CustomResourceDefinitions](#), to indicate the stability level of your CustomResourceDefinitions or advance your API to a new version with conversion between API representations. It also describes how to upgrade an object from one version to another.

Before you begin

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

- [KillerCoda](#)
- [Play with Kubernetes](#)

You should have an initial understanding of [custom resources](#).

Your Kubernetes server must be at or later than version v1.16. To check the version, enter `kubectl version`.

Overview

The CustomResourceDefinition API provides a workflow for introducing and upgrading to new versions of a CustomResourceDefinition.

When a CustomResourceDefinition is created, the first version is set in the `CustomResourceDefinition spec.versions` list to an appropriate stability level and a version number. For example `v1beta1` would indicate that the first version is not yet stable. All custom resource objects will initially be stored at this version.

Once the CustomResourceDefinition is created, clients may begin using the `v1beta1` API.

Later it might be necessary to add new version such as `v1`.

Adding a new version:

1. Pick a conversion strategy. Since custom resource objects need the ability to be served at both versions, that means they will sometimes be served in a different version than the one stored. To make this possible, the custom resource objects must sometimes be converted between the version they are stored at and the version they are served at. If the conversion involves schema changes and requires custom logic, a conversion webhook should be used. If there are no schema changes, the default `None` conversion strategy may be used and only the `apiVersion` field will be modified when serving different versions.
2. If using conversion webhooks, create and deploy the conversion webhook. See the [Webhook conversion](#) for more details.
3. Update the CustomResourceDefinition to include the new version in the `spec.versions` list with `served:true`. Also, set `spec.conversion` field to the selected conversion strategy. If using a conversion webhook, configure `spec.conversion.webhookClientConfig` field to call the webhook.

Once the new version is added, clients may incrementally migrate to the new version. It is perfectly safe for some clients to use the old version while others use the new version.

Migrate stored objects to the new version:

1. See the [upgrade existing objects to a new stored version](#) section.

It is safe for clients to use both the old and new version before, during and after upgrading the objects to a new stored version.

Removing an old version:

1. Ensure all clients are fully migrated to the new version. The kube-apiserver logs can be reviewed to help identify any clients that are still accessing via the old version.
2. Set `served` to `false` for the old version in the `spec.versions` list. If any clients are still unexpectedly using the old version they may begin reporting errors attempting to access the custom resource objects at the old version. If this occurs, switch back to using `served:true` on the old version, migrate the remaining clients to the new version and repeat this step.
3. Ensure the [upgrade of existing objects to the new stored version](#) step has been completed.
 1. Verify that the `storage` is set to `true` for the new version in the `spec.versions` list in the CustomResourceDefinition.
 2. Verify that the old version is no longer listed in the CustomResourceDefinition `status.storedVersions`.
4. Remove the old version from the CustomResourceDefinition `spec.versions` list.
5. Drop conversion support for the old version in conversion webhooks.

Specify multiple versions

The CustomResourceDefinition API `versions` field can be used to support multiple versions of custom resources that you have developed. Versions can have different schemas, and conversion webhooks can convert custom resources between versions. Webhook conversions should follow the [Kubernetes API conventions](#) wherever applicable. Specifically, See the [API change documentation](#) for a set of useful gotchas and suggestions.

Note: In `apiextensions.k8s.io/v1beta1`, there was a `version` field instead of `versions`. The `version` field is deprecated and optional, but if it is not empty, it must match the first item in the `versions` field.

This example shows a CustomResourceDefinition with two versions. For the first example, the assumption is all versions share the same schema with no conversion between them. The comments in the YAML provide more context.

[apiextensions.k8s.io/v1](#)

[apiextensions.k8s.io/v1beta1](#)

```
apiVersion: apiextensions.k8s.io/v1
kind: CustomResourceDefinition
metadata:
  # name must match the spec fields below, and be in the form: <plural>.<group>
  name: crontabs.example.com
spec:
  # group name to use for REST API: /apis/<group>/<version>
  group: example.com
  # list of versions supported by this CustomResourceDefinition
  versions:
    - name: v1beta1
      # Each version can be enabled/disabled by Served flag.
      served: true
      # One and only one version must be marked as the storage version.
      storage: true
      # A schema is required
      schema:
        openAPIV3Schema:
          type: object
          properties:
            host:
              type: string
            port:
              type: string
    - name: v1
      served: true
      storage: false
      schema:
        openAPIV3Schema:
          type: object
          properties:
            host:
              type: string
            port:
              type: string
  # The conversion section is introduced in Kubernetes 1.13+ with a default value of
  # None conversion (strategy sub-field set to None).
  conversion:
    # None conversion assumes the same schema for all versions and only sets the api
    # field of custom resources to the proper value
    strategy: None
    # either Namespaced or Cluster
    scope: Namespaced
  names:
    # plural name to be used in the URL: /apis/<group>/<version>/<plural>
    plural: crontabs
    # singular name to be used as an alias on the CLI and for display
    singular: crontab
    # kind is normally the CamelCased singular type. Your resource manifests use this
    kind: CronTab
    # shortNames allow shorter string to match your resource on the CLI
    shortNames:
      - ct
```

You can save the CustomResourceDefinition in a YAML file, then use `kubectl apply` to create it.

```
kubectl apply -f my-versioned-crontab.yaml
```

After creation, the API server starts to serve each enabled version at an HTTP REST endpoint. In the above example, the API versions are available at `/apis/example.com/v1beta1` and `/apis/example.com/v1`.

Version priority

Regardless of the order in which versions are defined in a CustomResourceDefinition, the version with the highest priority is used by kubectl as the default version to access objects. The priority is determined by parsing the `name` field to determine the version number, the stability (GA, Beta, or Alpha), and the sequence within that stability level.

The algorithm used for sorting the versions is designed to sort versions in the same way that the Kubernetes project sorts Kubernetes versions. Versions start with a `v` followed by a number, an optional `beta` or `alpha` designation, and optional additional numeric versioning information. Broadly, a version string might look like `v2` or `v2beta1`. Versions are sorted using the following algorithm:

- Entries that follow Kubernetes version patterns are sorted before those that do not.
- For entries that follow Kubernetes version patterns, the numeric portions of the version string is sorted largest to smallest.
- If the strings `beta` or `alpha` follow the first numeric portion, they are sorted in that order, after the equivalent string without the `beta` or `alpha` suffix (which is presumed to be the GA version).
- If another number follows the `beta`, or `alpha`, those numbers are also sorted from largest to smallest.
- Strings that don't fit the above format are sorted alphabetically and the numeric portions are not treated specially. Notice that in the example below, `foo1` is sorted above `foo10`. This is different from the sorting of the numeric portion of entries that do follow the Kubernetes version patterns.

This might make sense if you look at the following sorted version list:

```
- v10
- v2
- v1
- v11beta2
- v10beta3
- v3beta1
- v12alpha1
- v11alpha2
- foo1
- foo10
```

For the example in [Specify multiple versions](#), the version sort order is `v1`, followed by `v1beta1`. This causes the kubectl command to use `v1` as the default version unless the provided object specifies the version.

Version deprecation

FEATURE STATE: [Kubernetes v1.19 \[stable\]](#)

Starting in v1.19, a CustomResourceDefinition can indicate a particular version of the resource it defines is deprecated. When API requests to a deprecated version of that resource are made, a warning message is returned in the API response as a header. The warning message for each deprecated version of the resource can be customized if desired.

A customized warning message should indicate the deprecated API group, version, and kind, and should indicate what API group, version, and kind should be used instead, if applicable.

[apiextensions.k8s.io/v1](#)

[apiextensions.k8s.io/v1beta1](#)

```
apiVersion: apiextensions.k8s.io/v1
kind: CustomResourceDefinition
  name: crontabs.example.com
spec:
  group: example.com
  names:
    plural: crontabs
    singular: crontab
    kind: CronTab
  scope: Namespaced
  versions:
    - name: v1alpha1
      served: true
      storage: false
      # This indicates the v1alpha1 version of the custom resource is deprecated.
      # API requests to this version receive a warning header in the server response.
      deprecated: true
      # This overrides the default warning returned to API clients making v1alpha1 API
      deprecationWarning: "example.com/v1alpha1 CronTab is deprecated; see http://exam
        schema: ...
    - name: v1beta1
      served: true
      # This indicates the v1beta1 version of the custom resource is deprecated.
      # API requests to this version receive a warning header in the server response.
      # A default warning message is returned for this version.
      deprecated: true
      schema: ...
    - name: v1
      served: true
      storage: true
      schema: ...
```

Version removal

An older API version cannot be dropped from a CustomResourceDefinition manifest until existing stored data has been migrated to the newer API version for all clusters that served the older version of the custom resource, and the old version is removed from the `status.storedVersions` of the CustomResourceDefinition.

```
apiVersion: apiextensions.k8s.io/v1
kind: CustomResourceDefinition
name: crontabs.example.com
spec:
  group: example.com
  names:
    plural: crontabs
    singular: crontab
    kind: CronTab
  scope: Namespaced
  versions:
    - name: v1beta1
      # This indicates the v1beta1 version of the custom resource is no
      # API requests to this version receive a not found error in the se
      served: false
      schema: ...
    - name: v1
      served: true
      # The new served version should be set as the storage version
      storage: true
      schema: ...
```

Webhook conversion

FEATURE STATE: **Kubernetes v1.16 [stable]**

Note: Webhook conversion is available as beta since 1.15, and as alpha since Kubernetes 1.13. The [CustomResourceWebhookConversion](#) feature must be enabled, which is the case automatically for many clusters for beta features. Please refer to the [feature gate](#) documentation for more information.

The above example has a None conversion between versions which only sets the `apiVersion` field on conversion and does not change the rest of the object. The API server also supports webhook conversions that call an external service in case a conversion is required. For example when:

- custom resource is requested in a different version than stored version.
- Watch is created in one version but the changed object is stored in another version.
- custom resource PUT request is in a different version than storage version.

To cover all of these cases and to optimize conversion by the API server, the conversion requests may contain multiple objects in order to minimize the external calls. The webhook should perform these conversions independently.

Write a conversion webhook server

Please refer to the implementation of the [custom resource conversion webhook server](#) that is validated in a Kubernetes e2e test. The webhook handles the `ConversionReview` requests sent by the API servers, and sends back conversion results wrapped in `ConversionResponse`. Note that the request contains a list of custom resources that need to be converted independently without changing the order of objects. The example server

is organized in a way to be reused for other conversions. Most of the common code are located in the [framework file](#) that leaves only [one function](#) to be implemented for different conversions.

Note: The example conversion webhook server leaves the `ClientAuth` field [empty](#), which defaults to `NoClientCert`. This means that the webhook server does not authenticate the identity of the clients, supposedly API servers. If you need mutual TLS or other ways to authenticate the clients, see how to [authenticate API servers](#).

Permissible mutations

A conversion webhook must not mutate anything inside of `metadata` of the converted object other than `labels` and `annotations`. Attempted changes to `name`, `UID` and `namespace` are rejected and fail the request which caused the conversion. All other changes are ignored.

Deploy the conversion webhook service

Documentation for deploying the conversion webhook is the same as for the [admission webhook example service](#). The assumption for next sections is that the conversion webhook server is deployed to a service named `example-conversion-webhook-server` in `default` namespace and serving traffic on path `/crdconvert`.

Note: When the webhook server is deployed into the Kubernetes cluster as a service, it has to be exposed via a service on port 443 (The server itself can have an arbitrary port but the service object should map it to port 443). The communication between the API server and the webhook service may fail if a different port is used for the service.

Configure CustomResourceDefinition to use conversion webhooks

The `None` conversion example can be extended to use the conversion webhook by modifying `conversion` section of the `spec`:

[apiextensions.k8s.io/v1](#)

[apiextensions.k8s.io/v1beta1](#)

```
apiVersion: apiextensions.k8s.io/v1
kind: CustomResourceDefinition
metadata:
  # name must match the spec fields below, and be in the form: <plural>.<group>
  name: crontabs.example.com
spec:
  # group name to use for REST API: /apis/<group>/<version>
  group: example.com
  # list of versions supported by this CustomResourceDefinition
  versions:
    - name: v1beta1
      # Each version can be enabled/disabled by Served flag.
      served: true
      # One and only one version must be marked as the storage version.
      storage: true
      # Each version can define its own schema when there is no top-level
      # schema is defined.
      schema:
        openAPIV3Schema:
          type: object
          properties:
            hostPort:
              type: string
    - name: v1
      served: true
      storage: false
      schema:
        openAPIV3Schema:
          type: object
          properties:
            host:
              type: string
            port:
              type: string
  conversion:
    # a Webhook strategy instruct API server to call an external webhook for any conversion
    strategy: Webhook
    # webhook is required when strategy is `Webhook` and it configures the webhook endpoint
    webhook:
      # conversionReviewVersions indicates what ConversionReview versions are understood
      # The first version in the list understood by the API server is sent to the webhook
      # The webhook must respond with a ConversionReview object in the same version
      conversionReviewVersions: ["v1", "v1beta1"]
      clientConfig:
        service:
          namespace: default
          name: example-conversion-webhook-server
          path: /crdconvert
          caBundle: "Ci0tLS0tQk...<base64-encoded PEM bundle>...tLS0K"
    # 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
```

You can save the CustomResourceDefinition in a YAML file, then use `kubectl apply` to apply it.

```
kubectl apply -f my-versioned-crontab-with-conversion.yaml
```

Make sure the conversion service is up and running before applying new changes.

Contacting the webhook

Once the API server has determined a request should be sent to a conversion webhook, it needs to know how to contact the webhook. This is specified in the `webhookClientConfig` stanza of the webhook configuration.

Conversion webhooks can either be called via a URL or a service reference, and can optionally include a custom CA bundle to use to verify the TLS connection.

URL

`url` gives the location of the webhook, in standard URL form (`scheme://host:port/path`).

The `host` should not refer to a service running in the cluster; use a service reference by specifying the `service` field instead. The host might be resolved via external DNS in some apiservers (i.e., `kube-apiserver` cannot resolve in-cluster DNS as that would be a layering violation). `host` may also be an IP address.

Please note that using `localhost` or `127.0.0.1` as a `host` is risky unless you take great care to run this webhook on all hosts which run an apiserver which might need to make calls to this webhook. Such installations are likely to be non-portable or not readily run in a new cluster.

The scheme must be "https"; the URL must begin with "https://".

Attempting to use a user or basic auth (for example "user:password@") is not allowed. Fragments ("#...") and query parameters ("?...") are also not allowed.

Here is an example of a conversion webhook configured to call a URL (and expects the TLS certificate to be verified using system trust roots, so does not specify a `caBundle`):

[apiextensions.k8s.io/v1](#)

[apiextensions.k8s.io/v1beta1](#)

```
apiVersion: apiextensions.k8s.io/v1
kind: CustomResourceDefinition
...
spec:
  ...
  conversion:
    strategy: Webhook
    webhook:
      clientConfig:
        url: "https://my-webhook.example.com:9443/my-webhook-path"
      ...

```

Service Reference

The `service` stanza inside `webhookClientConfig` is a reference to the service for a conversion webhook. If the webhook is running within the cluster, then you should use `service` instead of `url`. The service

namespace and name are required. The port is optional and defaults to 443. The path is optional and defaults to "/".

Here is an example of a webhook that is configured to call a service on port "1234" at the subpath "/my-path", and to verify the TLS connection against the ServerName `my-service-name.my-service-namespace.svc` using a custom CA bundle.

[apiextensions.k8s.io/v1](https://kubernetes.io/docs/api-extensions/) [apiextensions.k8s.io/v1beta1](https://kubernetes.io/docs/api-extensions/)

[apiextensions.k8s.io/v1beta1](#)

```
apiVersion: apiextensions.k8s.io/v1
kind: CustomResourceDefinition
...
spec:
  ...
  conversion:
    strategy: Webhook
    webhook:
      clientConfig:
        service:
          namespace: my-service-namespace
          name: my-service-name
          path: /my-path
          port: 1234
      caBundle: "Ci0tLS0tQk...<base64-encoded PEM bundle>...tLS0K"
...

```

Webhook request and response

Request

Webhooks are sent a POST request, with `Content-Type: application/json`, with a `ConversionReview` API object in the `apiextensions.k8s.io` API group serialized to JSON as the body.

Webhooks can specify what versions of `ConversionReview` objects they accept with the `conversionReviewVersions` field in their `CustomResourceDefinition`:

[apiextensions.k8s.io/v1](https://kubernetes.io/docs/api-reference/ext-apis/) [apiextensions.k8s.io/v1beta1](https://kubernetes.io/docs/api-reference/ext-apis/)

apiextensions.k8s.io/v1beta1

```
apiVersion: apiextensions.k8s.io/v1
kind: CustomResourceDefinition
...
spec:
  ...
  conversion:
    strategy: Webhook
    webhook:
      conversionReviewVersions: ["v1", "v1beta1"]
```

`conversionReviewVersions` is a required field when creating `apiextensions.k8s.io/v1` custom resource definitions. Webhooks are required to support at least one `ConversionReview` version understood by the current and previous API server.

API servers send the first `ConversionReview` version in the `conversionReviewVersions` list they support. If none of the versions in the list are supported by the API server, the custom resource definition will not be allowed to be created. If an API server encounters a conversion

webhook configuration that was previously created and does not support any of the `ConversionReview` versions the API server knows how to send, attempts to call to the webhook will fail.

This example shows the data contained in an `ConversionReview` object for a request to convert `CronTab` objects to `example.com/v1`:

apiextensions.k8s.io/v1	apiextensions.k8s.io/v1beta1
---	--

```
{
  "apiVersion": "apiextensions.k8s.io/v1",
  "kind": "ConversionReview",
  "request": {
    # Random uid uniquely identifying this conversion call
    "uid": "705ab4f5-6393-11e8-b7cc-42010a800002",

    # The API group and version the objects should be converted to
    "desiredAPIVersion": "example.com/v1",

    # The list of objects to convert.
    # May contain one or more objects, in one or more versions.
    "objects": [
      {
        "kind": "CronTab",
        "apiVersion": "example.com/v1beta1",
        "metadata": {
          "creationTimestamp": "2019-09-04T14:03:02Z",
          "name": "local-crontab",
          "namespace": "default",
          "resourceVersion": "143",
          "uid": "3415a7fc-162b-4300-b5da-fd6083580d66"
        },
        "hostPort": "localhost:1234"
      },
      {
        "kind": "CronTab",
        "apiVersion": "example.com/v1beta1",
        "metadata": {
          "creationTimestamp": "2019-09-03T13:02:01Z",
          "name": "remote-crontab",
          "resourceVersion": "12893",
          "uid": "359a83ec-b575-460d-b553-d859cedde8a0"
        },
        "hostPort": "example.com:2345"
      }
    ]
  }
}
```

Response

Webhooks respond with a 200 HTTP status code, `Content-Type: application/json`, and a body containing a `ConversionReview` object (in the same version they were sent), with the `response` stanza populated, serialized to JSON.

If conversion succeeds, a webhook should return a `response` stanza containing the following fields:

- `uid`, copied from the `request.uid` sent to the webhook
- `result`, set to `{"status": "Success"}`
- `convertedObjects`, containing all of the objects from `request.objects`, converted to `request.desiredVersion`

Example of a minimal successful response from a webhook:

```
apiextensions.k8s.io/v1 apiextensions.k8s.io/v1beta1
```

```
{
  "apiVersion": "apiextensions.k8s.io/v1",
  "kind": "ConversionReview",
  "response": {
    # must match <request.uid>
    "uid": "705ab4f5-6393-11e8-b7cc-42010a800002",
    "result": {
      "status": "Success"
    },
    # Objects must match the order of request.objects, and have apiVersion set to <r
    # kind, metadata.uid, metadata.name, and metadata.namespace fields must not be c
    # metadata.labels and metadata.annotations fields may be changed by the webhook.
    # All other changes to metadata fields by the webhook are ignored.
    "convertedObjects": [
      {
        "kind": "CronTab",
        "apiVersion": "example.com/v1",
        "metadata": {
          "creationTimestamp": "2019-09-04T14:03:02Z",
          "name": "local-crontab",
          "namespace": "default",
          "resourceVersion": "143",
          "uid": "3415a7fc-162b-4300-b5da-fd6083580d66"
        },
        "host": "localhost",
        "port": "1234"
      },
      {
        "kind": "CronTab",
        "apiVersion": "example.com/v1",
        "metadata": {
          "creationTimestamp": "2019-09-03T13:02:01Z",
          "name": "remote-crontab",
          "resourceVersion": "12893",
          "uid": "359a83ec-b575-460d-b553-d859cedde8a0"
        },
        "host": "example.com",
        "port": "2345"
      }
    ]
  }
}
```

If conversion fails, a webhook should return a `response` stanza containing the following fields:

- `uid`, copied from the `request.uid` sent to the webhook
- `result`, set to `{"status": "Failed"}`

Warning: Failing conversion can disrupt read and write access to the custom resources, including the ability to update or delete the resources. Conversion failures should be avoided whenever possible, and should not be used to enforce validation constraints (use validation schemas or webhook admission instead).

Example of a response from a webhook indicating a conversion request failed, with an optional message:

[apiextensions.k8s.io/v1](#)[apiextensions.k8s.io/v1beta1](#)

```
{
  "apiVersion": "apiextensions.k8s.io/v1",
  "kind": "ConversionReview",
  "response": {
    "uid": "<value from request.uid>",
    "result": {
      "status": "Failed",
      "message": "hostPort could not be parsed into a separate host and port"
    }
  }
}
```

Writing, reading, and updating versioned CustomResourceDefinition objects

When an object is written, it is stored at the version designated as the storage version at the time of the write. If the storage version changes, existing objects are never converted automatically. However, newly-created or updated objects are written at the new storage version. It is possible for an object to have been written at a version that is no longer served.

When you read an object, you specify the version as part of the path. You can request an object at any version that is currently served. If you specify a version that is different from the object's stored version, Kubernetes returns the object to you at the version you requested, but the stored object is not changed on disk.

What happens to the object that is being returned while serving the read request depends on what is specified in the CRD's `spec.conversion`:

- if the default `strategy` value `None` is specified, the only modifications to the object are changing the `apiVersion` string and perhaps [pruning unknown fields](#) (depending on the configuration). Note that this is unlikely to lead to good results if the schemas differ between the storage and requested version. In particular, you should not use this strategy if the same data is represented in different fields between versions.
- if [webhook conversion](#) is specified, then this mechanism controls the conversion.

If you update an existing object, it is rewritten at the version that is currently the storage version. This is the only way that objects can change from one version to another.

To illustrate this, consider the following hypothetical series of events:

1. The storage version is `v1beta1`. You create an object. It is stored at version `v1beta1`
2. You add version `v1` to your CustomResourceDefinition and designate it as the storage version. Here the schemas for `v1` and `v1beta1` are identical, which is typically the case when promoting an API to stable in the Kubernetes ecosystem.
3. You read your object at version `v1beta1`, then you read the object again at version `v1`. Both returned objects are identical except for the `apiVersion` field.

4. You create a new object. It is stored at version `v1`. You now have two objects, one of which is at `v1beta1`, and the other of which is at `v1`.
5. You update the first object. It is now stored at version `v1` since that is the current storage version.

Previous storage versions

The API server records each version which has ever been marked as the storage version in the status field `storedVersions`. Objects may have been stored at any version that has ever been designated as a storage version. No objects can exist in storage at a version that has never been a storage version.

Upgrade existing objects to a new stored version

When deprecating versions and dropping support, select a storage upgrade procedure.

Option 1: Use the Storage Version Migrator

1. Run the [storage Version migrator](#)
2. Remove the old version from the CustomResourceDefinition `status.storedVersions` field.

Option 2: Manually upgrade the existing objects to a new stored version

The following is an example procedure to upgrade from `v1beta1` to `v1`.

1. Set `v1` as the storage in the CustomResourceDefinition file and apply it using kubectl. The `storedVersions` is now `v1beta1, v1`.
2. Write an upgrade procedure to list all existing objects and write them with the same content. This forces the backend to write objects in the current storage version, which is `v1`.
3. Remove `v1beta1` from the CustomResourceDefinition `status.storedVersions` field.

Note:

The flag `--subresource` is used with the kubectl get, patch, edit, and replace commands to fetch and update the subresources, `status` and `scale`, for all the API resources that support them. This flag is available starting from kubectl version v1.24. Previously, reading subresources (like `status`) via kubectl involved using `kubectl --raw`, and updating subresources using kubectl was not possible at all. Starting from v1.24, the `kubectl` tool can be used to edit or patch the `status` subresource on a CRD object. See [How to patch a Deployment using the subresource flag](#).

This page is part of the documentation for Kubernetes v1.28. If you are running a different version of Kubernetes, consult the documentation for that release.

Here is an example of how to patch the `status` subresource for a CRD object using `kubectl`:

```
kubectl patch customresourcedefinitions <CRD_Name> --subresource='
```

11.3 - Set up an Extension API Server

Setting up an extension API server to work with 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

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

- [Killercoda](#)
- [Play with Kubernetes](#)

To check the version, enter `kubectl version`.

- You must [configure the aggregation layer](#) and enable the apiserver flags.

Set up 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 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. When run, kubectl should return "No resources found.". This message indicates that everything worked but you currently have no objects of that resource type created.

What's next

- Walk through the steps to [configure the API 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](#).

11.4 - 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. Moreover, 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

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

- [Killercoda](#)
- [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, you can use the default scheduler (kube-scheduler) as your second scheduler. Clone the [Kubernetes source code from GitHub](#) and build the source.

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

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

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

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](#). Alternatively you can also use the [docker hub](#). For more details refer to the docker hub [documentation](#).

```
docker build -t gcr.io/my-gcp-project/my-kube-scheduler:1.0 .      # This command builds the Docker image
gcloud docker -- push gcr.io/my-gcp-project/my-kube-scheduler:1.0 # uses the Docker daemon provided by gcloud
```

Define a Kubernetes Deployment for the scheduler

Now that you have your scheduler in a container image, create a pod configuration for it and run it in your Kubernetes cluster. But instead of creating a pod directly in the cluster, you can 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`:

[admin/sched/my-scheduler.yaml](#) 

```
apiVersion: v1
kind: ServiceAccount
metadata:
  name: my-scheduler
  namespace: kube-system
---
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRoleBinding
metadata:
  name: my-scheduler-as-kube-scheduler
subjects:
- kind: ServiceAccount
  name: my-scheduler
  namespace: kube-system
roleRef:
  kind: ClusterRole
  name: system:kube-scheduler
  apiGroup: rbac.authorization.k8s.io
---
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRoleBinding
metadata:
  name: my-scheduler-as-volume-scheduler
subjects:
- kind: ServiceAccount
  name: my-scheduler
  namespace: kube-system
roleRef:
  kind: ClusterRole
  name: system:volume-scheduler
  apiGroup: rbac.authorization.k8s.io
---
apiVersion: rbac.authorization.k8s.io/v1
kind: RoleBinding
metadata:
  name: my-scheduler-extension-apiserver-authentication-reader
  namespace: kube-system
roleRef:
  kind: Role
  name: extension-apiserver-authentication-reader
  apiGroup: rbac.authorization.k8s.io
subjects:
- kind: ServiceAccount
  name: my-scheduler
  namespace: kube-system
---
apiVersion: v1
kind: ConfigMap
metadata:
  name: my-scheduler-config
  namespace: kube-system
data:
  my-scheduler-config.yaml: |
    apiVersion: kubescheduler.config.k8s.io/v1beta2
    kind: KubeSchedulerConfiguration
    profiles:
      - schedulerName: my-scheduler
    LeaderElection:
      leaderElect: false
---
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
            - --config=/etc/kubernetes/my-scheduler/my-scheduler-config.yaml
          image: gcr.io/my-gcp-project/my-kube-scheduler:1.0
          livenessProbe:
            httpGet:
              path: /healthz
              port: 10259
              scheme: HTTPS
            initialDelaySeconds: 15
          name: kube-second-scheduler
          readinessProbe:
            httpGet:
              path: /healthz
              port: 10259
              scheme: HTTPS
          resources:
            requests:
              cpu: '0.1'
          securityContext:
            privileged: false
          volumeMounts:
            - name: config-volume
              mountPath: /etc/kubernetes/my-scheduler
        hostNetwork: false
        hostPID: false
        volumes:
          - name: config-volume
            configMap:
              name: my-scheduler-config
```

In the above manifest, you use a [KubeSchedulerConfiguration](#) to customize the behavior of your scheduler implementation. This configuration has been passed to the `kube-scheduler` during initialization with the `--config` option. The `my-scheduler-config` ConfigMap stores the configuration file. The Pod of the `my-scheduler` Deployment mounts the `my-scheduler-config` ConfigMap as a volume.

In the aforementioned Scheduler Configuration, your scheduler implementation is represented via a [KubeSchedulerProfile](#).

Note: To determine if a scheduler is responsible for scheduling a specific Pod, the `spec.schedulerName` field in a PodTemplate or Pod manifest must match the `schedulerName` field of the

KubeSchedulerProfile. All schedulers running in the cluster must have unique names.

Also, note that you create a dedicated service account `my-scheduler` and bind the ClusterRole `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 and [Scheduler Configuration reference](#) for detailed description of other customizable `kube-scheduler` configurations.

Run the second scheduler in the cluster

In order to run your scheduler in a Kubernetes cluster, 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	REST
....			
my-scheduler-lnf4s-4744f	1/1	Running	0
...			

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

Enable leader election

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

Update the following fields for the KubeSchedulerConfiguration in the `my-scheduler-config` ConfigMap in your YAML file:

- `leaderElection.leaderElect` to `true`
- `leaderElection.resourceNamespace` to `<lock-object-namespace>`
- `leaderElection.resourceName` to `<lock-object-name>`

Note: The control plane creates the lock objects for you, but the namespace must already exist. You can use the `kube-system` namespace.

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` and `leases` resources, as in the following example:

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

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

```
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:
  - coordination.k8s.io
  resources:
  - leases
  verbs:
  - create
- apiGroups:
  - coordination.k8s.io
  resourceNames:
  - kube-scheduler
  - my-scheduler
  resources:
  - leases
  verbs:
  - get
  - update
- apiGroups:
  - ""
  resourceNames:
  - kube-scheduler
  - my-scheduler
  resources:
  - endpoints
  verbs:
  - delete
  - get
  - patch
  - update
```

Specify schedulers for pods

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

- Pod spec without any scheduler name

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

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

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

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

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

- Pod spec with `default-scheduler`

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

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

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

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

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

- Pod spec with `my-scheduler`

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

```
apiVersion: v1
kind: Pod
metadata:
  name: annotation-second-scheduler
  labels:
    name: multischeduler-example
spec:
  schedulerName: my-scheduler
  containers:
    - name: pod-with-second-annotation-container
      image: registry.k8s.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 for the scheduler in the `schedulerName` field of the mapping `KubeSchedulerProfile`.

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, you can look at the "Scheduled" entries in the event logs to verify that the pods were scheduled by the desired schedulers.

```
kubectl get events
```

You can also use a [custom scheduler configuration](#) or a custom container image for the cluster's main scheduler by modifying its static pod manifest on the relevant control plane nodes.

11.5 - 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

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

- [Killercoda](#)
- [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 create deployment node-hello --image=gcr.io/google-samples/node-hello:latest
```

Using `kubectl` to start a proxy server

This command starts a proxy to the Kubernetes API server:

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

Exploring the Kubernetes API

When the proxy server is running, you can explore the API using `curl`, `wget`, or a browser.

Get the API versions:

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

The output should look similar to this:

```
{
  "kind": "APIVersions",
  "versions": [
    "v1"
  ],
  "serverAddressByClientCIDRs": [
    {
      "clientCIDR": "0.0.0.0/0",
      "serverAddress": "10.0.2.15:8443"
    }
  ]
}
```

Get a list of pods:

```
curl http://localhost:8080/api/v1/namespaces/default/pods
```

The output should look similar to this:

```
{
  "kind": "PodList",
  "apiVersion": "v1",
  "metadata": {
    "resourceVersion": "33074"
  },
  "items": [
    {
      "metadata": {
        "name": "kubernetes-bootcamp-2321272333-ix8pt",
        "generateName": "kubernetes-bootcamp-2321272333-",
        "namespace": "default",
        "uid": "ba21457c-6b1d-11e6-85f7-1ef9f1dab92b",
        "resourceVersion": "33003",
        "creationTimestamp": "2016-08-25T23:43:30Z",
        "labels": {
          "pod-template-hash": "2321272333",
          "run": "kubernetes-bootcamp"
        },
        ...
      }
    }
```

What's next

Learn more about [kubectl proxy](#).

11.6 - Use a SOCKS5 Proxy to Access the Kubernetes API

FEATURE STATE: [Kubernetes v1.24 \[stable\]](#)

This page shows how to use a SOCKS5 proxy to access the API of a remote Kubernetes cluster. This is useful when the cluster you want to access does not expose its API directly on the public internet.

Before you begin

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

- [Killercoda](#)
- [Play with Kubernetes](#)

Your Kubernetes server must be at or later than version v1.24. To check the version, enter `kubectl version`.

You need SSH client software (the `ssh` tool), and an SSH service running on the remote server. You must be able to log in to the SSH service on the remote server.

Task context

Note: This example tunnels traffic using SSH, with the SSH client and server acting as a SOCKS proxy. You can instead use any other kind of [SOCKS5](#) proxies.

Figure 1 represents what you're going to achieve in this task.

- You have a client computer, referred to as local in the steps ahead, from where you're going to create requests to talk to the Kubernetes API.
- The Kubernetes server/API is hosted on a remote server.
- You will use SSH client and server software to create a secure SOCKS5 tunnel between the local and the remote server. The HTTPS traffic between the client and the Kubernetes API will flow over the SOCKS5 tunnel, which is itself tunneled over SSH.

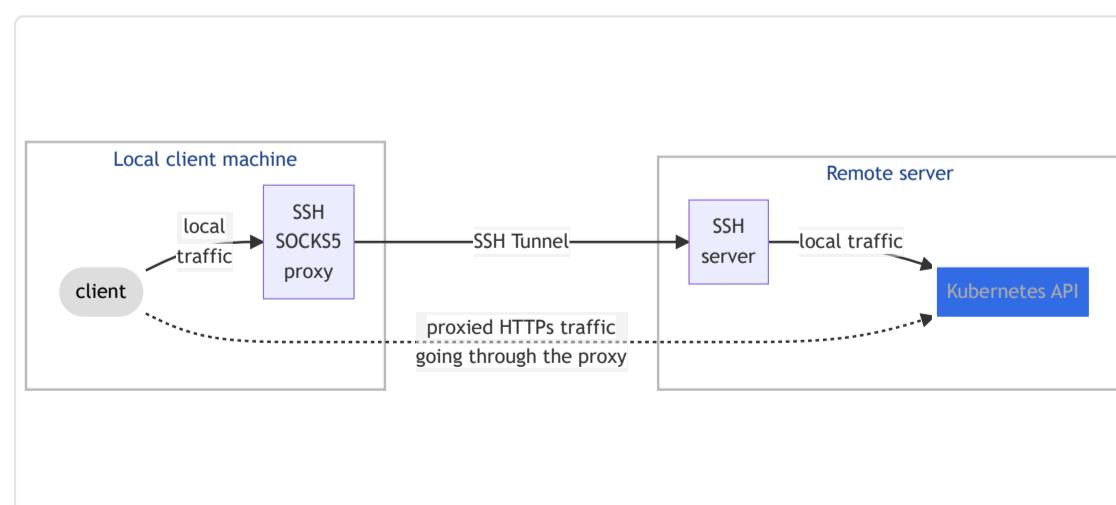


Figure 1. SOCKS5 tutorial components

Using ssh to create a SOCKS5 proxy

The following command starts a SOCKS5 proxy between your client machine and the remote SOCKS server:

```
# The SSH tunnel continues running in the foreground after you run this command
ssh -D 1080 -q -N username@kubernetes-remote-server.example
```

The SOCKS5 proxy lets you connect to your cluster's API server based on the following configuration:

- `-D 1080` : opens a SOCKS proxy on local port :1080.
- `-q` : quiet mode. Causes most warning and diagnostic messages to be suppressed.
- `-N` : Do not execute a remote command. Useful for just forwarding ports.
- `username@kubernetes-remote-server.example` : the remote SSH server behind which the Kubernetes cluster is running (eg: a bastion host).

Client configuration

To access the Kubernetes API server through the proxy you must instruct `kubectl` to send queries through the `socks` proxy we created earlier. Do this by either setting the appropriate environment variable, or via the `proxy-url` attribute in the `kubeconfig` file. Using an environment variable:

```
export HTTPS_PROXY=socks5://localhost:1080
```

To always use this setting on a specific `kubectl` context, specify the `proxy-url` attribute in the relevant `cluster` entry within the `~/.kube/config` file. For example:

```
apiVersion: v1
clusters:
- cluster:
  certificate-authority-data: LRMEMMW2 # shortened for readability
  server: https://<API_SERVER_IP_ADDRESS>:6443 # the "Kubernetes API"
  proxy-url: socks5://localhost:1080 # the "SSH SOCKS5 proxy" in this guide
  name: default
contexts:
- context:
  cluster: default
  user: default
  name: default
current-context: default
kind: Config
preferences: {}
users:
- name: default
  user:
    client-certificate-data: LS0tLS1CR== # shortened for readability
    client-key-data: LS0tLS1CRUdJT= # shortened for readability
```

Once you have created the tunnel via the ssh command mentioned earlier, and defined either the environment variable or the `proxy-url` attribute, you can interact with your cluster through that proxy. For example:

```
kubectl get pods
```

NAMESPACE	NAME	READY	STATUS
kube-system	coredns-85cb69466-k1wq8	1/1	Running

Note:

- Before `kubectl 1.24`, most `kubectl` commands worked when using a socks proxy, except `kubectl exec`.
- `kubectl` supports both `HTTPS_PROXY` and `https_proxy` environment variables. These are used by other programs that support SOCKS, such as `curl`. Therefore in some cases it will be better to define the environment variable on the command line:

```
HTTPS_PROXY=socks5://localhost:1080 kubectl get pods
```

- When using `proxy-url`, the proxy is used only for the relevant `kubectl` context, whereas the environment variable will affect all contexts.
- The k8s API server hostname can be further protected from DNS leakage by using the `socks5h` protocol name instead of the more commonly known `socks5` protocol shown above. In this case, `kubectl` will ask the proxy server (such as an ssh bastion) to resolve the k8s API server domain name, instead of resolving it on the system running `kubectl`. Note also that with `socks5h`, a k8s API server URL like `https://localhost:6443/api` does not refer to your local client computer. Instead, it refers to `localhost` as known on the proxy server (eg the ssh bastion).

Clean up

Stop the ssh port-forwarding process by pressing `CTRL+C` on the terminal where it is running.

Type `unset https_proxy` in a terminal to stop forwarding http traffic through the proxy.

Further reading

- [OpenSSH remote login client](#)

11.7 - Set up Konnectivity service

The Konnectivity service provides a TCP level proxy for the control plane to cluster communication.

Before you begin

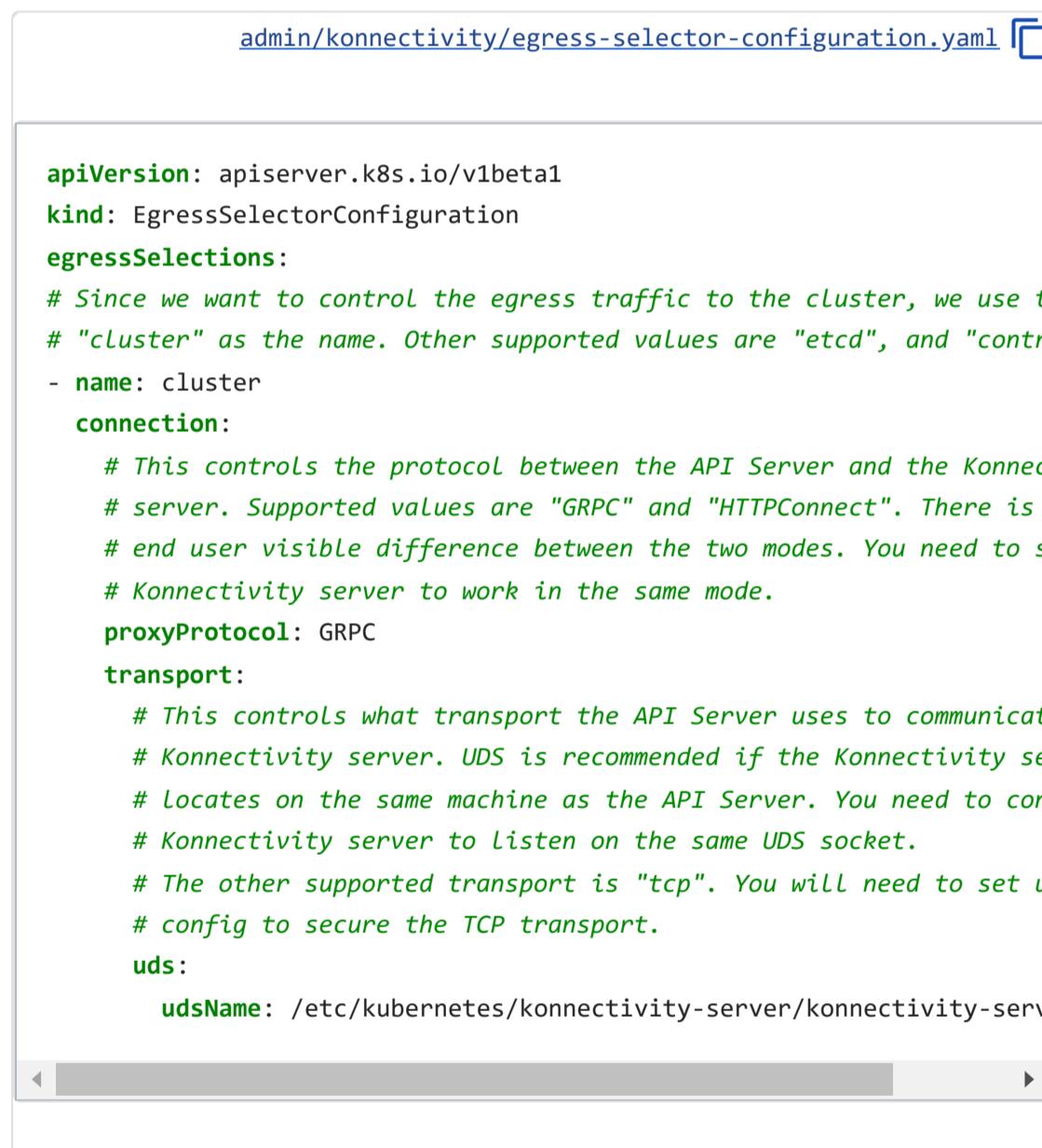
You need to have a Kubernetes cluster, and the `kubectl` command-line tool must be configured to communicate with your cluster. It is recommended to run this tutorial on a cluster with at least two nodes that are not acting as control plane hosts. If you do not already have a cluster, you can create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

Configure the Konnectivity service

The following steps require an egress configuration, for example:

```
admin/konnectivity/egress-selector-configuration.yaml ⚙️
```

```
apiVersion: apiserver.k8s.io/v1beta1
kind: EgressSelectorConfiguration
egressSelections:
  # Since we want to control the egress traffic to the cluster, we use t
  # "cluster" as the name. Other supported values are "etcd", and "contr
  - name: cluster
    connection:
      # This controls the protocol between the API Server and the Konne
      # server. Supported values are "GRPC" and "HTTPConnect". There is
      # end user visible difference between the two modes. You need to s
      # Konnectivity server to work in the same mode.
      proxyProtocol: GRPC
    transport:
      # This controls what transport the API Server uses to communicat
      # Konnectivity server. UDS is recommended if the Konnectivity se
      # Locates on the same machine as the API Server. You need to con
      # Konnectivity server to listen on the same UDS socket.
      # The other supported transport is "tcp". You will need to set u
      # config to secure the TCP transport.
      uds:
        udsName: /etc/kubernetes/konnectivity-server/konnectivity-serv
```



You need to configure the API Server to use the Konnectivity service and direct the network traffic to the cluster nodes:

1. Make sure that [Service Account Token Volume Projection](#) feature enabled in your cluster. It is enabled by default since Kubernetes v1.20.
2. Create an egress configuration file such as `admin/konnectivity/egress-selector-configuration.yaml`.
3. Set the `--egress-selector-config-file` flag of the API Server to the path of your API Server egress configuration file.
4. If you use UDS connection, add volumes config to the kube-apiserver:

```
spec:  
  containers:  
    volumeMounts:  
      - name: konnectivity-uds  
        mountPath: /etc/kubernetes/konnectivity-server  
        readOnly: false  
    volumes:  
      - name: konnectivity-uds  
        hostPath:  
          path: /etc/kubernetes/konnectivity-server  
          type: DirectoryOrCreate
```

Generate or obtain a certificate and kubeconfig for konnectivity-server. For example, you can use the OpenSSL command line tool to issue a X.509 certificate, using the cluster CA certificate `/etc/kubernetes/pki/ca.crt` from a control-plane host.

```
openssl req -subj "/CN=system:konnectivity-server" -new -newkey rsa:2048  
openssl x509 -req -in konnectivity.csr -CA /etc/kubernetes/pki/ca.crt  
SERVER=$(kubectl config view -o jsonpath='{.clusters..server}')  
kubectl --kubeconfig /etc/kubernetes/konnectivity-server.conf config set identity.kubeconfig /etc/kubernetes/konnectivity-server.conf  
kubectl --kubeconfig /etc/kubernetes/konnectivity-server.conf config set identity.caFile /etc/kubernetes/pki/ca.crt  
kubectl --kubeconfig /etc/kubernetes/konnectivity-server.conf config set identity.keyFile /etc/kubernetes/konnectivity.key  
rm -f konnectivity.crt konnectivity.key konnectivity.csr
```

Next, you need to deploy the Konnectivity server and agents. [kubernetes-sigs/apiserver-network-proxy](#) is a reference implementation.

Deploy the Konnectivity server on your control plane node. The provided `konnectivity-server.yaml` manifest assumes that the Kubernetes components are deployed as a static Pod in your cluster. If not, you can deploy the Konnectivity server as a DaemonSet.

[admin/konnectivity/konnectivity-server.yaml](#) 

```
apiVersion: v1
kind: Pod
metadata:
  name: konnectivity-server
  namespace: kube-system
spec:
  priorityClassName: system-cluster-critical
  hostNetwork: true
  containers:
    - name: konnectivity-server-container
      image: registry.k8s.io/kas-network-proxy/proxy-server:v0.0.37
      command: ["/proxy-server"]
      args: [
        "--logtostderr=true",
        # This needs to be consistent with the value set in egress
        "--uds-name=/etc/kubernetes/konnectivity-server/konnectivity-uds",
        "--delete-existing-uds-file",
        # The following two lines assume the Konnectivity server is
        # deployed on the same machine as the apiserver, and the cert
        # key of the API Server are at the specified Location.
        "--cluster-cert=/etc/kubernetes/pki/apiserver.crt",
        "--cluster-key=/etc/kubernetes/pki/apiserver.key",
        # This needs to be consistent with the value set in egress
        "--mode=grpc",
        "--server-port=0",
        "--agent-port=8132",
        "--admin-port=8133",
        "--health-port=8134",
        "--agent-namespace=kube-system",
        "--agent-service-account=konnectivity-agent",
        "--kubeconfig=/etc/kubernetes/konnectivity-server.conf",
        "--authentication-audience=system:konnectivity-server"
      ]
      livenessProbe:
        httpGet:
          scheme: HTTP
          host: 127.0.0.1
          port: 8134
          path: /healthz
        initialDelaySeconds: 30
        timeoutSeconds: 60
      ports:
        - name: agentport
          containerPort: 8132
          hostPort: 8132
        - name: adminport
          containerPort: 8133
          hostPort: 8133
        - name: healthport
          containerPort: 8134
          hostPort: 8134
      volumeMounts:
        - name: k8s-certs
          mountPath: /etc/kubernetes/pki
          readOnly: true
        - name: kubeconfig
          mountPath: /etc/kubernetes/konnectivity-server.conf
          readOnly: true
        - name: konnectivity-uds
          mountPath: /etc/kubernetes/konnectivity-server
          readOnly: false
      volumes:
        - name: k8s-certs
          hostPath:
            path: /etc/kubernetes/pki
        - name: kubeconfig
```

```
hostPath:  
  path: /etc/kubernetes/konnectivity-server.conf  
  type: FileOrCreate  
- name: konnectivity-uds  
  hostPath:  
    path: /etc/kubernetes/konnectivity-server  
    type: DirectoryOrCreate
```

Then deploy the Konnectivity agents in your cluster:

[admin/konnectivity/konnectivity-agent.yaml](#) 

```

apiVersion: apps/v1
# Alternatively, you can deploy the agents as Deployments. It is not necessary
# to have an agent on each node.
kind: DaemonSet
metadata:
  labels:
    addonmanager.kubernetes.io/mode: Reconcile
    k8s-app: konnectivity-agent
  namespace: kube-system
  name: konnectivity-agent
spec:
  selector:
    matchLabels:
      k8s-app: konnectivity-agent
  template:
    metadata:
      labels:
        k8s-app: konnectivity-agent
    spec:
      priorityClassName: system-cluster-critical
      tolerations:
        - key: "CriticalAddonsOnly"
          operator: "Exists"
      containers:
        - image: us.gcr.io/k8s-artifacts-prod/kas-network-proxy/proxy-
          name: konnectivity-agent
          command: ["/proxy-agent"]
          args: [
            "--logtostderr=true",
            "--ca-cert=/var/run/secrets/kubernetes.io/serviceaccount/ca.crt",
            # Since the konnectivity server runs with hostNetwork=true,
            # this is the IP address of the master machine.
            "--proxy-server-host=35.225.206.7",
            "--proxy-server-port=8132",
            "--admin-server-port=8133",
            "--health-server-port=8134",
            "--service-account-token-path=/var/run/secrets/token",
          ]
      volumeMounts:
        - mountPath: /var/run/secrets/tokens
          name: konnectivity-agent-token
      livenessProbe:
        httpGet:
          port: 8134
          path: /healthz
        initialDelaySeconds: 15
        timeoutSeconds: 15
      serviceAccountName: konnectivity-agent
      volumes:
        - name: konnectivity-agent-token
      projected:
        sources:
          - serviceAccountToken:
              path: konnectivity-agent-token
              audience: system:konnectivity-server

```

Last, if RBAC is enabled in your cluster, create the relevant RBAC rules:

[admin/konnectivity/konnectivity-rbac.yaml](#) 

```
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRoleBinding
metadata:
  name: system:konnectivity-server
  labels:
    kubernetes.io/cluster-service: "true"
    addonmanager.kubernetes.io/mode: Reconcile
roleRef:
  apiGroup: rbac.authorization.k8s.io
  kind: ClusterRole
  name: system:auth-delegator
subjects:
  - apiGroup: rbac.authorization.k8s.io
    kind: User
    name: system:konnectivity-server
---
apiVersion: v1
kind: ServiceAccount
metadata:
  name: konnectivity-agent
  namespace: kube-system
  labels:
    kubernetes.io/cluster-service: "true"
    addonmanager.kubernetes.io/mode: Reconcile
```

12 - TLS

Understand how to protect traffic within your cluster using Transport Layer Security (TLS).

12.1 - Configure Certificate Rotation for the Kubelet

This page shows how to enable and configure certificate rotation for the kubelet.

FEATURE STATE: [Kubernetes v1.19 \[stable\]](#)

Before you begin

- Kubernetes version 1.8.0 or later is required

Overview

The kubelet uses certificates for authenticating to the Kubernetes API. By default, these certificates are issued with one year expiration so that they do not need to be renewed too frequently.

Kubernetes contains [kubelet certificate rotation](#), 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.

The `kube-controller-manager` process accepts an argument `--cluster-signing-duration` (`--experimental-cluster-signing-duration` prior to 1.19) 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 `--cluster-signing-duration` parameter, and the signed certificate will be attached to the certificate signing request.

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. This can happen at any point between 30% and 10% of the time remaining on the certificate. 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.

12.2 - Manage TLS Certificates in a Cluster

Kubernetes provides a `certificates.k8s.io` API, which lets you provision TLS certificates signed by a Certificate Authority (CA) that you control. These CA and certificates can be used by your workloads to establish trust.

`certificates.k8s.io` API uses a protocol that is similar to the [ACME draft](#).

Note: Certificates created using the `certificates.k8s.io` API are signed by a [dedicated CA](#). It is possible to configure your cluster to use the cluster root CA for this purpose, but you should never rely on this. Do not assume that these certificates will validate against the cluster root CA.

Before you begin

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

- [Killercoda](#)
- [Play with Kubernetes](#)

You need the `cfssl` tool. You can download `cfssl` from <https://github.com/cloudflare/cfssl/releases>.

Some steps in this page use the `jq` tool. If you don't have `jq`, you can install it via your operating system's software sources, or fetch it from <https://jqlang.github.io/jq/>.

Trusting TLS in a cluster

Trusting the [custom CA](#) from an application running as a pod usually requires some extra application configuration. You will need to add the CA certificate bundle to the list of CA certificates that the TLS client or server trusts. For example, you would do this with a golang TLS config by parsing the certificate chain and adding the parsed certificates to the `RootCAs` field in the [`tls.Config`](#) struct.

Note:

Even though the custom CA certificate may be included in the filesystem (in the ConfigMap `kube-root-ca.crt`), you should not use that certificate authority for any purpose other than to verify internal Kubernetes endpoints. An example of an internal Kubernetes endpoint is the Service named `kubernetes` in the default namespace.

If you want to use a custom certificate authority for your workloads, you should generate that CA separately, and distribute its CA certificate using a [ConfigMap](#) that your pods have access to read.

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.

Create a certificate signing request

Generate a private key and certificate signing request (or CSR) by running the following command:

```
cat <<EOF | cfssl genkey - | cfssljson -bare server
{
  "hosts": [
    "my-svc.my-namespace.svc.cluster.local",
    "my-pod.my-namespace.pod.cluster.local",
    "192.0.2.24",
    "10.0.34.2"
  ],
  "CN": "my-pod.my-namespace.pod.cluster.local",
  "key": {
    "algo": "ecdsa",
    "size": 256
  }
}
EOF
```

Where `192.0.2.24` is the service's cluster IP, `my-svc.my-namespace.svc.cluster.local` is the service's DNS name, `10.0.34.2` is the pod's IP and `my-pod.my-namespace.pod.cluster.local` is the pod's DNS name. You should see the output similar to:

```
2022/02/01 11:45:32 [INFO] generate received request
2022/02/01 11:45:32 [INFO] received CSR
2022/02/01 11:45:32 [INFO] generating key: ecdsa-256
2022/02/01 11:45:32 [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 CertificateSigningRequest object to send to the Kubernetes API

Generate a CSR manifest (in YAML), and send it to the API server. You can do that by running the following command:

```
cat <<EOF | kubectl apply -f -
apiVersion: certificates.k8s.io/v1
kind: CertificateSigningRequest
metadata:
  name: my-svc.my-namespace
spec:
  request: $(cat server.csr | base64 | tr -d '\n')
  signerName: example.com/serving
  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. You are also requesting a certificate with the "digital signature", "key encipherment", and "server auth" key usages, signed by an example `example.com/serving` signer. A specific `signerName` must be requested. View documentation for [supported signer names](#) for more information.

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, 01 Feb 2022 11:49:15 -0500
Requesting User: yourname@example.com
Signer:         example.com/serving
Status:         Pending
Subject:
  Common Name:  my-pod.my-namespace.pod.cluster.local
  Serial Number:
Subject Alternative Names:
  DNS Names:   my-pod.my-namespace.pod.cluster.local
                 my-svc.my-namespace.svc.cluster.local
  IP Addresses: 192.0.2.24
                 10.0.34.2
Events: <none>
```

Get the CertificateSigningRequest approved

Approving the [certificate signing request](#) is either done by an automated approval process or on a one off basis by a cluster administrator. If you're authorized to approve a certificate request, you can do that manually using `kubectl` ; for example:

```
kubectl certificate approve my-svc.my-namespace
```

```
certificatesigningrequest.certificates.k8s.io/my-svc.my-namespace appro
```

You should now see the following:

```
kubectl get csr
```

NAME	AGE	SIGNERNAME	REQUESTOR
my-svc.my-namespace	10m	example.com/serving	yourname@example.com

This means the certificate request has been approved and is waiting for the requested signer to sign it.

Sign the CertificateSigningRequest

Next, you'll play the part of a certificate signer, issue the certificate, and upload it to the API.

A signer would typically watch the CertificateSigningRequest API for objects with its `signerName`, check that they have been approved, sign certificates for those requests, and update the API object status with the issued certificate.

Create a Certificate Authority

You need an authority to provide the digital signature on the new certificate.

First, create a signing certificate by running the following:

```
cat <<EOF | cfssl gencert -initca - | cfssljson -bare ca
{
  "CN": "My Example Signer",
  "key": {
    "algo": "rsa",
    "size": 2048
  }
}
EOF
```

You should see output similar to:

```
2022/02/01 11:50:39 [INFO] generating a new CA key and certificate from
2022/02/01 11:50:39 [INFO] generate received request
2022/02/01 11:50:39 [INFO] received CSR
2022/02/01 11:50:39 [INFO] generating key: rsa-2048
2022/02/01 11:50:39 [INFO] encoded CSR
2022/02/01 11:50:39 [INFO] signed certificate with serial number 263983
```

This produces a certificate authority key file (`ca-key.pem`) and certificate (`ca.pem`).

Issue a certificate

[tls/server-signing-config.json](#)

```
{  
  "signing": {  
    "default": {  
      "usages": [  
        "digital signature",  
        "key encipherment",  
        "server auth"  
      ],  
      "expiry": "876000h",  
      "ca_constraint": {  
        "is_ca": false  
      }  
    }  
  }  
}
```

Use a `server-signing-config.json` signing configuration and the certificate authority key file and certificate to sign the certificate request:

```
kubectl get csr my-svc.my-namespace -o jsonpath='{.spec.request}' | \  
base64 --decode | \  
cfssl sign -ca ca.pem -ca-key ca-key.pem -config server-signing-conf:  
cfssljson -bare ca-signed-server
```

You should see the output similar to:

```
2022/02/01 11:52:26 [INFO] signed certificate with serial number 576048
```

This produces a signed serving certificate file, `ca-signed-server.pem`.

Upload the signed certificate

Finally, populate the signed certificate in the API object's status:

```
kubectl get csr my-svc.my-namespace -o json | \  
jq '.status.certificate = "'$(base64 ca-signed-server.pem | tr -d '\\' \  
kubectl replace --raw /apis/certificates.k8s.io/v1/certificatesigning
```

Note: This uses the command line tool `jq` to populate the base64-encoded content in the `.status.certificate` field. If you do not have `jq`, you can also save the JSON output to a file, populate this field manually, and upload the resulting file.

Once the CSR is approved and the signed certificate is uploaded, run:

```
kubectl get csr
```

The output is similar to:

NAME	AGE	SIGNERNAME	REQUESTOR
my-svc.my-namespace	20m	example.com/serving	yourname@example.com

Download the certificate and use it

Now, as the requesting user, you can download the issued certificate and save it to a `server.crt` file by running the following:

```
kubectl get csr my-svc.my-namespace -o jsonpath='{.status.certificate}'  
| base64 --decode > server.crt
```

Now you can populate `server.crt` and `server-key.pem` in a [Secret](#) that you could later mount into a Pod (for example, to use with a webserver that serves HTTPS).

```
kubectl create secret tls server --cert server.crt --key server-key.pem
```

```
secret/server created
```

Finally, you can populate `ca.pem` into a [ConfigMap](#) and use it as the trust root to verify the serving certificate:

```
kubectl create configmap example-serving-ca --from-file ca.crt=ca.pem
```

```
configmap/example-serving-ca created
```

Approving CertificateSigningRequests

A Kubernetes administrator (with appropriate permissions) can manually approve (or deny) CertificateSigningRequests 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.

Caution:

The ability to approve CSRs decides who trusts whom within your environment. The ability to approve CSRs should not be granted broadly or lightly.

You should make sure that you confidently understand both the verification requirements that fall on the approver **and** the repercussions of issuing a specific certificate before you grant the `approve` permission.

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.

For more information on certificate approval and access control, read the [Certificate Signing Requests](#) reference page.

Configuring your cluster to provide signing

This page assumes that a signer is set up 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.

12.3 - Manual Rotation of CA Certificates

This page shows how to manually rotate the certificate authority (CA) certificates.

Before you begin

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

- [Killercoda](#)
- [Play with Kubernetes](#)
- For more information about authentication in Kubernetes, see [Authenticating](#).
- For more information about best practices for CA certificates, see [Single root CA](#).

Rotate the CA certificates manually

Caution:

Make sure to back up your certificate directory along with configuration files and any other necessary files.

This approach assumes operation of the Kubernetes control plane in a HA configuration with multiple API servers. Graceful termination of the API server is also assumed so clients can cleanly disconnect from one API server and reconnect to another.

Configurations with a single API server will experience unavailability while the API server is being restarted.

1. Distribute the new CA certificates and private keys (for example:

`ca.crt` , `ca.key` , `front-proxy-ca.crt` , and `front-proxy-ca.key`) to all your control plane nodes in the Kubernetes certificates directory.

2. Update the `--root-ca-file` flag for the [kube-controller-manager](#) to include both old and new CA, then restart the `kube-controller-manager`.

Any [ServiceAccount](#) created after this point will get Secrets that include both old and new CAs.

Note:

The files specified by the `kube-controller-manager` flags `--client-ca-file` and `--cluster-signing-cert-file` cannot be CA bundles. If these flags and `--root-ca-file` point to the same `ca.crt` file which is now a bundle (includes both old and new CA) you will face an error. To workaround this problem you can copy the new CA to a separate file and make the flags `--client-ca-`

file and --cluster-signing-cert-file point to the copy. Once ca.crt is no longer a bundle you can restore the problem flags to point to ca.crt and delete the copy.

[Issue 1350](#) for kubeadm tracks an bug with the kube-controller-manager being unable to accept a CA bundle.

3. Wait for the controller manager to update ca.crt in the service account Secrets to include both old and new CA certificates.

If any Pods are started before new CA is used by API servers, the new Pods get this update and will trust both old and new CAs.

4. Restart all pods using in-cluster configurations (for example: kube-proxy, CoreDNS, etc) so they can use the updated certificate authority data from Secrets that link to ServiceAccounts.

- o Make sure CoreDNS, kube-proxy and other Pods using in-cluster configurations are working as expected.

5. Append the both old and new CA to the file against --client-ca-file and --kubelet-certificate-authority flag in the kube-apiserver configuration.

6. Append the both old and new CA to the file against --client-ca-file flag in the kube-scheduler configuration.

7. Update certificates for user accounts by replacing the content of client-certificate-data and client-key-data respectively.

For information about creating certificates for individual user accounts, see [Configure certificates for user accounts](#).

Additionally, update the certificate-authority-data section in the kubeconfig files, respectively with Base64-encoded old and new certificate authority data

8. Update the --root-ca-file flag for the Cloud Controller Manager to include both old and new CA, then restart the cloud-controller-manager.

Note: If your cluster does not have a cloud-controller-manager, you can skip this step.

9. Follow the steps below in a rolling fashion.

1. Restart any other aggregated API servers or webhook handlers to trust the new CA certificates.

2. Restart the kubelet by update the file against clientCAFile in kubelet configuration and certificate-authority-data in kubelet.conf to use both the old and new CA on all nodes.

If your kubelet is not using client certificate rotation, update client-certificate-data and client-key-data in kubelet.conf on all nodes along with the kubelet client certificate file usually found in /var/lib/kubelet/pki .

3. Restart API servers with the certificates (apiserver.crt , apiserver-kubelet-client.crt and front-proxy-client.crt) signed by new CA. You can use the existing private keys or new private keys. If you changed the private keys then update these in the Kubernetes certificates directory as well.

Since the Pods in your cluster trust both old and new CAs, there will be a momentarily disconnection after which pods' Kubernetes clients reconnect to the new API server. The new API

server uses a certificate signed by the new CA.

- Restart the `kube-scheduler` to use and trust the new CAs.
- Make sure control plane components logs no TLS errors.

Note: To generate certificates and private keys for your cluster using the `openssl` command line tool, see [Certificates \(openssl\)](#). You can also use [cfssl](#).

4. Annotate any DaemonSets and Deployments to trigger pod replacement in a safer rolling fashion.

```
for namespace in $(kubectl get namespace -o jsonpath='{.items[*]}'  
    for name in $(kubectl get deployments -n $namespace -o jsonpath='{.items[*].name}')  
        kubectl patch deployment -n ${namespace} ${name} -p '{"spec":  
            "strategy": {"rollingUpdate": {"maxUnavailable": 1}}}  
    done  
    for name in $(kubectl get daemonset -n $namespace -o jsonpath='{.items[*].name}')  
        kubectl patch daemonset -n ${namespace} ${name} -p '{"spec":  
            "strategy": {"rollingUpdate": {"maxUnavailable": 1}}}  
    done  
done
```

Note: To limit the number of concurrent disruptions that your application experiences, see [configure pod disruption budget](#).

Depending on how you use StatefulSets you may also need to perform a rolling update of the StatefulSets.

10. If your cluster is using bootstrap tokens to join nodes, update the `ConfigMap cluster-info` in the `kube-public` namespace with new CA.

```
base64_encoded_ca=$(base64 -w0 /etc/kubernetes/pki/ca.crt)  
  
kubectl get cm/cluster-info --namespace kube-public -o yaml | \  
  /bin/sed "s/^(certificate-authority-data:[\t ]*)/\1 ${base64_encoded_ca}" |   
  kubectl apply -f -
```

11. Verify the cluster functionality.

1. Check the logs from control plane components, along with the `kubelet` and the `kube-proxy`. Ensure those components are not reporting any TLS errors; see [looking at the logs](#) for more details.
2. Validate logs from any aggregated api servers and pods using in-cluster config.

12. Once the cluster functionality is successfully verified:

1. Update all service account tokens to include new CA certificate only.
 - All pods using an in-cluster `kubeconfig` will eventually need to be restarted to pick up the new Secret, so that no Pods are relying on the old cluster CA.
2. Restart the control plane components by removing the old CA from the `kubeconfig` files and the files against `--client-ca-file`, `--root-ca-file` flags resp.

3. On each node, restart the kubelet by removing the old CA from file against the `clientCAFile` flag and from the kubelet kubeconfig file. You should carry this out as a rolling update.

If your cluster lets you make this change, you can also roll it out by replacing nodes rather than reconfiguring them.

13 - Manage Cluster Daemons

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

13.1 - Perform a Rolling Update on a DaemonSet

This page shows how to perform a rolling update on a DaemonSet.

Before you begin

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

- [Killercoda](#)
- [Play with Kubernetes](#)

DaemonSet Update Strategy

DaemonSet has two update strategy types:

- `OnDelete` : With `OnDelete` update strategy, after you update a DaemonSet template, new DaemonSet pods will *only* be created when you manually delete old DaemonSet pods. This is the same behavior of DaemonSet in Kubernetes version 1.5 or before.
- `RollingUpdate` : This is the default update strategy. With `RollingUpdate` update strategy, after you update a DaemonSet template, old DaemonSet pods will be killed, and new DaemonSet pods will be created automatically, in a controlled fashion. At most one pod of the DaemonSet will be running on each node during the whole update process.

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), `.spec.minReadySeconds` (default to 0) and `.spec.updateStrategy.rollingUpdate.maxSurge` (defaults to 0) as well.

Creating a DaemonSet with `RollingUpdate` update strategy

This YAML file specifies a DaemonSet with an update strategy as 'RollingUpdate'

[controllers/fluentd-daemonset.yaml](#) 

```
apiVersion: apps/v1
kind: DaemonSet
metadata:
  name: fluentd-elasticsearch
  namespace: kube-system
  labels:
    k8s-app: fluentd-logging
spec:
  selector:
    matchLabels:
      name: fluentd-elasticsearch
  updateStrategy:
    type: RollingUpdate
    rollingUpdate:
      maxUnavailable: 1
  template:
    metadata:
      labels:
        name: fluentd-elasticsearch
    spec:
      tolerations:
        # these tolerations are to have the daemonset runnable on control plane
        # remove them if your control plane nodes should not run pods
        - key: node-role.kubernetes.io/control-plane
          operator: Exists
          effect: NoSchedule
        - key: node-role.kubernetes.io/master
          operator: Exists
          effect: NoSchedule
      containers:
        - name: fluentd-elasticsearch
          image: quay.io/fluentd_elasticsearch/fluentd:v2.5.2
          volumeMounts:
            - name: varlog
              mountPath: /var/log
            - name: varlibdockercontainers
              mountPath: /var/lib/docker/containers
              readOnly: true
          terminationGracePeriodSeconds: 30
      volumes:
        - name: varlog
          hostPath:
            path: /var/log
        - name: varlibdockercontainers
          hostPath:
            path: /var/lib/docker/containers
```

After verifying the update strategy of the DaemonSet manifest, create the DaemonSet:

```
kubectl create -f https://k8s.io/examples/controllers/fluentd-daemonset
```

Alternatively, use `kubectl apply` to create the same DaemonSet if you plan to update the DaemonSet with `kubectl apply`.

```
kubectl apply -f https://k8s.io/examples/controllers/fluentd-daemonset
```

Checking DaemonSet **RollingUpdate** update strategy

Check the update strategy of your DaemonSet, and make sure it's set to

`RollingUpdate` :

```
kubectl get ds/fluentd-elasticsearch -o go-template='{{.spec.updateStrat
```

If you haven't created the DaemonSet in the system, check your DaemonSet manifest with the following command instead:

```
kubectl apply -f https://k8s.io/examples/controllers/fluentd-daemonset
```

The output from both commands should be:

```
RollingUpdate
```

If the output isn't `RollingUpdate`, go back and modify the DaemonSet object or manifest accordingly.

Updating a DaemonSet template

Any updates to a `RollingUpdate` DaemonSet `.spec.template` will trigger a rolling update. Let's update the DaemonSet by applying a new YAML file. This can be done with several different `kubectl` commands.

[controllers/fluentd-daemonset-update.yaml](#) 

```
apiVersion: apps/v1
kind: DaemonSet
metadata:
  name: fluentd-elasticsearch
  namespace: kube-system
  labels:
    k8s-app: fluentd-logging
spec:
  selector:
    matchLabels:
      name: fluentd-elasticsearch
  updateStrategy:
    type: RollingUpdate
    rollingUpdate:
      maxUnavailable: 1
  template:
    metadata:
      labels:
        name: fluentd-elasticsearch
    spec:
      tolerations:
        # these tolerations are to have the daemonset runnable on control plane
        # remove them if your control plane nodes should not run pods
        - key: node-role.kubernetes.io/control-plane
          operator: Exists
          effect: NoSchedule
        - key: node-role.kubernetes.io/master
          operator: Exists
          effect: NoSchedule
      containers:
        - name: fluentd-elasticsearch
          image: quay.io/fluentd_elasticsearch/fluentd:v2.5.2
          resources:
            limits:
              memory: 200Mi
            requests:
              cpu: 100m
              memory: 200Mi
            volumeMounts:
              - name: varlog
                mountPath: /var/log
              - name: varlibdockercontainers
                mountPath: /var/lib/docker/containers
                readOnly: true
            terminationGracePeriodSeconds: 30
          volumes:
            - name: varlog
              hostPath:
                path: /var/log
            - name: varlibdockercontainers
              hostPath:
                path: /var/lib/docker/containers
```

Declarative commands

If you update DaemonSets using [configuration files](#), use `kubectl apply`:

```
kubectl apply -f https://k8s.io/examples/controllers/fluentd-daemonset
```

Imperative commands

If you update DaemonSets using [imperative commands](#), use `kubectl edit`:

```
kubectl edit ds/fluentd-elasticsearch -n kube-system
```

Updating only the container image

If you only 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/fluentd-elasticsearch fluentd-elasticsearch=quay.
```

Watching the rolling update status

Finally, watch the rollout status of the latest DaemonSet rolling update:

```
kubectl rollout status ds/fluentd-elasticsearch -n kube-system
```

When the rollout is complete, the output is similar to this:

```
daemonset "fluentd-elasticsearch" 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 name=fluentd-elasticsearch -o wide -n kube-system
```

Once you've found those nodes, delete some non-DaemonSet pods from the node to make room for new DaemonSet pods.

Note: This will cause service disruption when deleted pods are not controlled by any controllers or pods are not replicated. This does not respect [PodDisruptionBudget](#) either.

Broken rollout

If the recent DaemonSet template update is broken, for example, the container is crash looping, or the container image doesn't exist (often due to a typo), DaemonSet rollout won't progress.

To fix this, 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.

Clean up

Delete DaemonSet from a namespace :

```
kubectl delete ds fluentd-elasticsearch -n kube-system
```

What's next

- See [Performing a rollback on a DaemonSet](#)
- See [Creating a DaemonSet to adopt existing DaemonSet pods](#)

13.2 - Perform a Rollback on a DaemonSet

This page shows how to perform a rollback on a [DaemonSet](#).

Before you begin

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

- [Killercoda](#)
- [Play with Kubernetes](#)

Your Kubernetes server must be at or later than version 1.7. To check the version, enter `kubectl version`.

You should already 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 only 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
```

Note: If **--to-revision** flag is not specified, kubectl picks the most recent revision.

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 inside the cluster control plane.

To watch the progress of the rollback:

```
kubectl rollout status ds/<daemonset-name>
```

When the rollback is complete, the output is similar to:

```
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.

To see what is stored in each revision, find the DaemonSet revision raw resources:

```
kubectl get controllerrevision -l <daemonset-selector-key>=<daemonset-
```

This returns a list of ControllerRevisions:

NAME	CONTROLLER	REVIS
<daemonset-name>-<revision-hash>	DaemonSet/<daemonset-name>	1
<daemonset-name>-<revision-hash>	DaemonSet/<daemonset-name>	2

Each ControllerRevision stores the annotations and template of a DaemonSet revision.

`kubectl rollout undo` takes a specific ControllerRevision and replaces DaemonSet template with the template stored in the ControllerRevision. `kubectl rollout undo` is equivalent to updating DaemonSet template to a previous revision through other commands, such as `kubectl edit` or `kubectl apply`.

Note: DaemonSet revisions only roll forward. That is to say, after a rollback completes, the revision number (`.revision` field) of the ControllerRevision being rolled back to will advance. For example, if you have revision 1 and 2 in the system, and roll back from revision 2 to revision 1, the ControllerRevision with `.revision: 1` will become `.revision: 3`.

Troubleshooting

- See [troubleshooting DaemonSet rolling update](#).

13.3 - Running Pods on Only Some Nodes

This page demonstrates how you can run Pods on only some Nodes as part of a DaemonSet

Before you begin

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

- [KillerCoda](#)
- [Play with Kubernetes](#)

Running Pods on only some Nodes

Imagine that you want to run a DaemonSet, but you only need to run those daemon pods on nodes that have local solid state (SSD) storage. For example, the Pod might provide cache service to the node, and the cache is only useful when low-latency local storage is available.

Step 1: Add labels to your nodes

Add the label `ssd=true` to the nodes which have SSDs.

```
kubectl label nodes example-node-1 example-node-2 ssd=true
```

Step 2: Create the manifest

Let's create a DaemonSet which will provision the daemon pods on the SSD labeled nodes only.

Next, use a `nodeSelector` to ensure that the DaemonSet only runs Pods on nodes with the `ssd` label set to "true".

[controllers/daemonset-label-selector.yaml](#) 

```
apiVersion: apps/v1
kind: DaemonSet
metadata:
  name: ssd-driver
  labels:
    app: nginx
spec:
  selector:
    matchLabels:
      app: ssd-driver-pod
  template:
    metadata:
      labels:
        app: ssd-driver-pod
    spec:
      nodeSelector:
        ssd: "true"
      containers:
        - name: example-container
          image: example-image
```

Step 3: Create the DaemonSet

Create the DaemonSet from the manifest by using `kubectl create` or `kubectl apply`

Let's label another node as `ssd=true`.

```
kubectl label nodes example-node-3 ssd=true
```

Labelling the node automatically triggers the control plane (specifically, the DaemonSet controller) to run a new daemon pod on that node.

```
kubectl get pods -o wide
```

The output is similar to:

NAME	READY	STATUS	RESTARTS	AGE
<daemonset-name><some-hash-01>	1/1	Running	0	13s
<daemonset-name><some-hash-02>	1/1	Running	0	13s
<daemonset-name><some-hash-03>	1/1	Running	0	5s

14 - Networking

Learn how to configure networking for your cluster.

14.1 - Adding entries to Pod /etc/hosts with HostAliases

Adding entries to a Pod's `/etc/hosts` file provides Pod-level override of hostname resolution when DNS and other options are not applicable. You can add these custom entries with the `HostAliases` field in `PodSpec`.

Modification not using `HostAliases` is not suggested because the file is managed by the `kubelet` and can be overwritten on during Pod creation/restart.

Default hosts file content

Start an Nginx Pod which is assigned a Pod IP:

```
kubectl run nginx --image nginx
```

```
pod/nginx created
```

Examine a Pod IP:

```
kubectl get pods --output=wide
```

NAME	READY	STATUS	RESTARTS	AGE	IP	NODE
nginx	1/1	Running	0	13s	10.200.0.4	worker0

The hosts file content would look like this:

```
kubectl exec nginx -- cat /etc/hosts
```

```
# Kubernetes-managed hosts file.  
127.0.0.1      localhost  
::1      localhost ip6-localhost ip6-loopback  
fe00::0 ip6-localnet  
fe00::0 ip6-mcastprefix  
fe00::1 ip6-allnodes  
fe00::2 ip6-allrouters  
10.200.0.4      nginx
```

By default, the `hosts` file only includes IPv4 and IPv6 boilerplates like `localhost` and its own hostname.

Adding additional entries with hostAliases

In addition to the default boilerplate, you can add additional entries to the `hosts` file. For example: to resolve `foo.local`, `bar.local` to `127.0.0.1` and `foo.remote`, `bar.remote` to `10.1.2.3`, you can configure `HostAliases` for a Pod under `.spec.hostAliases`:

```
service/networking/hostaliases-pod.yaml □
```

```
apiVersion: v1
kind: Pod
metadata:
  name: hostaliases-pod
spec:
  restartPolicy: Never
  hostAliases:
    - ip: "127.0.0.1"
      hostnames:
        - "foo.local"
        - "bar.local"
    - ip: "10.1.2.3"
      hostnames:
        - "foo.remote"
        - "bar.remote"
  containers:
    - name: cat-hosts
      image: busybox:1.28
      command:
        - cat
      args:
        - "/etc/hosts"
```

You can start a Pod with that configuration by running:

```
kubectl apply -f https://k8s.io/examples/service/networking/hostaliase
```

```
pod/hostaliases-pod created
```

Examine a Pod's details to see its IPv4 address and its status:

```
kubectl get pod --output=wide
```

NAME	READY	STATUS	RESTARTS	AGE
hostaliases-pod	0/1	Completed	0	6s

The `hosts` file content looks like this:

```
kubectl logs hostaliases-pod
```

```
# Kubernetes-managed hosts file.  
127.0.0.1      localhost  
::1      localhost ip6-localhost ip6-loopback  
fe00::0 ip6-localnet  
fe00::0 ip6-mcastprefix  
fe00::1 ip6-allnodes  
fe00::2 ip6-allrouters  
10.200.0.5      hostaliases-pod  
  
# Entries added by HostAliases.  
127.0.0.1      foo.local      bar.local  
10.1.2.3        foo.remote     bar.remote
```

with the additional entries specified at the bottom.

Why does the kubelet manage the hosts file?

The kubelet manages the `hosts` file for each container of the Pod to prevent the container runtime from modifying the file after the containers have already been started. Historically, Kubernetes always used Docker Engine as its container runtime, and Docker Engine would then modify the `/etc/hosts` file after each container had started.

Current Kubernetes can use a variety of container runtimes; even so, the kubelet manages the hosts file within each container so that the outcome is as intended regardless of which container runtime you use.

Caution:

Avoid making manual changes to the hosts file inside a container.

If you make manual changes to the hosts file, those changes are lost when the container exits.

14.2 - Validate IPv4/IPv6 dual-stack

This document shares how to validate IPv4/IPv6 dual-stack enabled Kubernetes clusters.

Before you begin

- Provider support for dual-stack networking (Cloud provider or otherwise must be able to provide Kubernetes nodes with routable IPv4/IPv6 network interfaces)
- A [network plugin](#) that supports dual-stack networking.
- [Dual-stack enabled](#) cluster

Your Kubernetes server must be at or later than version v1.23. To check the version, enter `kubectl version`.

Note: While you can validate with an earlier version, the feature is only GA and officially supported since v1.23.

Validate addressing

Validate node addressing

Each dual-stack Node should have a single IPv4 block and a single IPv6 block allocated. Validate that IPv4/IPv6 Pod address ranges are configured by running the following command. Replace the sample node name with a valid dual-stack Node from your cluster. In this example, the Node's name is `k8s-linuxpool1-34450317-0` :

```
kubectl get nodes k8s-linuxpool1-34450317-0 -o go-template --template=
```

```
10.244.1.0/24  
2001:db8::/64
```

There should be one IPv4 block and one IPv6 block allocated.

Validate that the node has an IPv4 and IPv6 interface detected. Replace node name with a valid node from the cluster. In this example the node name is `k8s-linuxpool1-34450317-0` :

```
kubectl get nodes k8s-linuxpool1-34450317-0 -o go-template --template=
```

```
Hostname: k8s-linuxpool1-34450317-0  
InternalIP: 10.0.0.5  
InternalIP: 2001:db8:10::5
```

Validate Pod addressing

Validate that a Pod has an IPv4 and IPv6 address assigned. Replace the Pod name with a valid Pod in your cluster. In this example the Pod name is `pod01`:

```
kubectl get pods pod01 -o go-template --template='{{range .status.podI
```

```
10.244.1.4  
2001:db8::4
```

You can also validate Pod IPs using the Downward API via the `status.podIPs` fieldPath. The following snippet demonstrates how you can expose the Pod IPs via an environment variable called `MY_POD_IPS` within a container.

```
env:  
- name: MY_POD_IPS  
  valueFrom:  
    fieldRef:  
      fieldPath: status.podIPs
```

The following command prints the value of the `MY_POD_IPS` environment variable from within a container. The value is a comma separated list that corresponds to the Pod's IPv4 and IPv6 addresses.

```
kubectl exec -it pod01 -- set | grep MY_POD_IPS
```

```
MY_POD_IPS=10.244.1.4,2001:db8::4
```

The Pod's IP addresses will also be written to `/etc/hosts` within a container. The following command executes a cat on `/etc/hosts` on a dual stack Pod. From the output you can verify both the IPv4 and IPv6 IP address for the Pod.

```
kubectl exec -it pod01 -- cat /etc/hosts
```

```
# Kubernetes-managed hosts file.  
127.0.0.1 localhost  
::1 localhost ip6-localhost ip6-loopback  
fe00::0 ip6-localnet  
fe00::0 ip6-mcastprefix  
fe00::1 ip6-allnodes  
fe00::2 ip6-allrouters  
10.244.1.4 pod01  
2001:db8::4 pod01
```

Validate Services

Create the following Service that does not explicitly define `.spec.ipFamilyPolicy`. Kubernetes will assign a cluster IP for the Service from the first configured `service-cluster-ip-range` and set the

```
.spec.ipFamilyPolicy to SingleStack .
```

[service/networking/dual-stack-default-svc.yaml](#)

```
apiVersion: v1
kind: Service
metadata:
  name: my-service
  labels:
    app.kubernetes.io/name: MyApp
spec:
  selector:
    app.kubernetes.io/name: MyApp
  ports:
    - protocol: TCP
      port: 80
```

Use `kubectl` to view the YAML for the Service.

```
kubectl get svc my-service -o yaml
```

The Service has `.spec.ipFamilyPolicy` set to `SingleStack` and `.spec.clusterIP` set to an IPv4 address from the first configured range set via `--service-cluster-ip-range` flag on `kube-controller-manager`.

```
apiVersion: v1
kind: Service
metadata:
  name: my-service
  namespace: default
spec:
  clusterIP: 10.0.217.164
  clusterIPs:
  - 10.0.217.164
  ipFamilies:
  - IPv4
  ipFamilyPolicy: SingleStack
  ports:
  - port: 80
    protocol: TCP
    targetPort: 9376
  selector:
    app.kubernetes.io/name: MyApp
  sessionAffinity: None
  type: ClusterIP
status:
  loadBalancer: {}
```

Create the following Service that explicitly defines `IPv6` as the first array element in `.spec.ipFamilies`. Kubernetes will assign a cluster IP for the Service from the IPv6 range configured `service-cluster-ip-range` and set the `.spec.ipFamilyPolicy` to `SingleStack`.

[service/networking/dual-stack-ipfamilies-ipv6.yaml](#)

```
apiVersion: v1
kind: Service
metadata:
  name: my-service
  labels:
    app.kubernetes.io/name: MyApp
spec:
  ipFamilies:
  - IPv6
  selector:
    app.kubernetes.io/name: MyApp
  ports:
  - protocol: TCP
    port: 80
```

Use `kubectl` to view the YAML for the Service.

```
kubectl get svc my-service -o yaml
```

The Service has `.spec.ipFamilyPolicy` set to `SingleStack` and `.spec.clusterIP` set to an IPv6 address from the IPv6 range set via `--service-cluster-ip-range` flag on `kube-controller-manager`.

```
apiVersion: v1
kind: Service
metadata:
  labels:
    app.kubernetes.io/name: MyApp
  name: my-service
spec:
  clusterIP: 2001:db8:fd00::5118
  clusterIPs:
  - 2001:db8:fd00::5118
  ipFamilies:
  - IPv6
  ipFamilyPolicy: SingleStack
  ports:
  - port: 80
    protocol: TCP
    targetPort: 80
  selector:
    app.kubernetes.io/name: MyApp
  sessionAffinity: None
  type: ClusterIP
status:
  loadBalancer: {}
```

Create the following Service that explicitly defines `PreferDualStack` in `.spec.ipFamilyPolicy`. Kubernetes will assign both IPv4 and IPv6 addresses (as this cluster has dual-stack enabled) and select the `.spec.ClusterIP` from the list of `.spec.ClusterIPs` based on the address family of the first element in the `.spec.ipFamilies` array.

[service/networking/dual-stack-preferred-svc.yaml](#) 

```
apiVersion: v1
kind: Service
metadata:
  name: my-service
  labels:
    app.kubernetes.io/name: MyApp
spec:
  ipFamilyPolicy: PreferDualStack
  selector:
    app.kubernetes.io/name: MyApp
  ports:
    - protocol: TCP
      port: 80
```

Note:

The `kubectl get svc` command will only show the primary IP in the `CLUSTER-IP` field.

```
kubectl get svc -l app.kubernetes.io/name=MyApp
```

NAME	TYPE	CLUSTER-IP	EXTERNAL-IP	PORT(S)	AG
my-service	ClusterIP	10.0.216.242	<none>	80/TCP	5s

Validate that the Service gets cluster IPs from the IPv4 and IPv6 address blocks using `kubectl describe`. You may then validate access to the service via the IPs and ports.

```
kubectl describe svc -l app.kubernetes.io/name=MyApp
```

Name:	my-service
Namespace:	default
Labels:	app.kubernetes.io/name=MyApp
Annotations:	<none>
Selector:	app.kubernetes.io/name=MyApp
Type:	ClusterIP
IP Family Policy:	PreferDualStack
IP Families:	IPv4,IPv6
IP:	10.0.216.242
IPs:	10.0.216.242,2001:db8:fd00::af55
Port:	<unset> 80/TCP
TargetPort:	9376/TCP
Endpoints:	<none>
Session Affinity:	None
Events:	<none>

Create a dual-stack load balanced Service

If the cloud provider supports the provisioning of IPv6 enabled external load balancers, create the following Service with `PreferDualStack` in `.spec.ipFamilyPolicy`, `IPv6` as the first element of the `.spec.ipFamilies` array and the `type` field set to `LoadBalancer`.

[service/networking/dual-stack-prefer-ipv6-lb-svc.yaml](#) 

```
apiVersion: v1
kind: Service
metadata:
  name: my-service
  labels:
    app.kubernetes.io/name: MyApp
spec:
  ipFamilyPolicy: PreferDualStack
  ipFamilies:
  - IPv6
  type: LoadBalancer
  selector:
    app.kubernetes.io/name: MyApp
  ports:
  - protocol: TCP
    port: 80
```

Check the Service:

```
kubectl get svc -l app.kubernetes.io/name=MyApp
```

Validate that the Service receives a `CLUSTER-IP` address from the IPv6 address block along with an `EXTERNAL-IP`. You may then validate access to the service via the IP and port.

NAME	TYPE	CLUSTER-IP	EXTERNAL-IP	PORTS
my-service	LoadBalancer	2001:db8:fd00::7ebc	2603:1030:805::5	80/TCP

15 - Extend kubectl with plugins

Extend kubectl by creating and installing kubectl plugins.

This guide demonstrates how to install and write extensions for [kubectl](#). By thinking of core `kubectl` commands as essential building blocks for interacting with a Kubernetes cluster, a cluster administrator can think of plugins as a means of utilizing these building blocks to create more complex behavior. Plugins extend `kubectl` with new sub-commands, allowing for new and custom features not included in the main distribution of `kubectl`.

Before you begin

You need to have a working `kubectl` binary installed.

Installing kubectl plugins

A plugin is a standalone executable file, whose name begins with `kubectl-`. To install a plugin, move its executable file to anywhere on your `PATH`.

You can also discover and install kubectl plugins available in the open source using [Krew](#). Krew is a plugin manager maintained by the Kubernetes SIG CLI community.

Caution: Kubectl plugins available via the Krew [plugin index](#) are not audited for security. You should install and run third-party plugins at your own risk, since they are arbitrary programs running on your machine.

Discovering plugins

`kubectl` provides a command `kubectl plugin list` that searches your `PATH` for valid plugin executables. Executing this command causes a traversal of all files in your `PATH`. Any files that are executable, and begin with `kubectl-` will show up *in the order in which they are present in your PATH* in this command's output. A warning will be included for any files beginning with `kubectl-` that are *not* executable. A warning will also be included for any valid plugin files that overlap each other's name.

You can use [Krew](#) to discover and install `kubectl` plugins from a community-curated [plugin index](#).

Limitations

It is currently not possible to create plugins that overwrite existing `kubectl` commands. For example, creating a plugin `kubectl-version` will cause that plugin to never be executed, as the existing `kubectl version` command will always take precedence over it. Due to this limitation, it is also *not* possible to use plugins to add new subcommands to existing `kubectl` commands. For example, adding a subcommand `kubectl create foo` by naming your plugin `kubectl-create-foo` will cause that plugin to be ignored.

`kubectl plugin list` shows warnings for any valid plugins that attempt to do this.

Writing kubectl plugins

You can write a plugin in any programming language or script that allows you to write command-line commands.

There is no plugin installation or pre-loading required. Plugin executables receive the inherited environment from the `kubectl` binary. A plugin determines which command path it wishes to implement based on its name. For example, a plugin named `kubectl-foo` provides a command `kubectl foo`. You must install the plugin executable somewhere in your `PATH`.

Example plugin

```
#!/bin/bash

# optional argument handling
if [[ "$1" == "version" ]]
then
    echo "1.0.0"
    exit 0
fi

# optional argument handling
if [[ "$1" == "config" ]]
then
    echo "$KUBECONFIG"
    exit 0
fi

echo "I am a plugin named kubectl-foo"
```

Using a plugin

To use a plugin, make the plugin executable:

```
sudo chmod +x ./kubectl-foo
```

and place it anywhere in your `PATH`:

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

You may now invoke your plugin as a `kubectl` command:

```
kubectl foo
```

```
I am a plugin named kubectl-foo
```

All args and flags are passed as-is to the executable:

```
kubectl foo version
```

1.0.0

All environment variables are also passed as-is to the executable:

```
export KUBECONFIG=~/.kube/config
kubectl foo config
```

```
/home/<user>/ .kube/config
```

```
KUBECONFIG=/etc/kube/config kubectl foo config
```

```
/etc/kube/config
```

Additionally, the first argument that is passed to a plugin will always be the full path to the location where it was invoked (`$0` would equal `/usr/local/bin/kubectl-foo` in the example above).

Naming a plugin

As seen in the example above, a plugin determines the command path that it will implement based on its filename. Every sub-command in the command path that a plugin targets, is separated by a dash (-). For example, a plugin that wishes to be invoked whenever the command `kubectl foo bar baz` is invoked by the user, would have the filename of `kubectl-foo-bar-baz` .

Flags and argument handling

Note:

The plugin mechanism does *not* create any custom, plugin-specific values or environment variables for a plugin process.

An older kubectl plugin mechanism provided environment variables such as `KUBECTL_PLUGINS_CURRENT_NAMESPACE` ; that no longer happens.

kubectl plugins must parse and validate all of the arguments passed to them. See [using the command line runtime package](#) for details of a Go library aimed at plugin authors.

Here are some additional cases where users invoke your plugin while providing additional flags and arguments. This builds upon the `kubectl-foo-bar-baz` plugin from the scenario above.

If you run `kubectl foo bar baz arg1 --flag=value arg2` , kubectl's plugin mechanism will first try to find the plugin with the longest possible name, which in this case would be `kubectl-foo-bar-baz-arg1` . Upon not finding that plugin, kubectl then treats the last dash-separated value as an argument (`arg1` in this case), and attempts to find the next longest possible name, `kubectl-foo-bar-baz` . Upon having found a plugin with this name, kubectl then invokes that plugin, passing all args and flags after the plugin's name as arguments to the plugin process.

Example:

```
# create a plugin
echo -e '#!/bin/bash\n\n'necho "My first command-line argument was $1"
sudo chmod +x ./kubectl-foo-bar-baz

# "install" your plugin by moving it to a directory in your $PATH
sudo mv ./kubectl-foo-bar-baz /usr/local/bin

# check that kubectl recognizes your plugin
kubectl plugin list
```

The following kubectl-compatible plugins are available:

```
/usr/local/bin/kubectl-foo-bar-baz
```

```
# test that calling your plugin via a "kubectl" command works
# even when additional arguments and flags are passed to your
# plugin executable by the user.
kubectl foo bar baz arg1 --meaningless-flag=true
```

```
My first command-line argument was arg1
```

As you can see, your plugin was found based on the `kubectl` command specified by a user, and all extra arguments and flags were passed as-is to the plugin executable once it was found.

Names with dashes and underscores

Although the `kubectl` plugin mechanism uses the dash (-) in plugin filenames to separate the sequence of sub-commands processed by the plugin, it is still possible to create a plugin command containing dashes in its commandline invocation by using underscores (_) in its filename.

Example:

```
# create a plugin containing an underscore in its filename
echo -e '#!/bin/bash\n\n'necho "I am a plugin with a dash in my name" >
sudo chmod +x ./kubectl-foo_bar

# move the plugin into your $PATH
sudo mv ./kubectl-foo_bar /usr/local/bin

# You can now invoke your plugin via kubectl:
kubectl foo-bar
```

```
I am a plugin with a dash in my name
```

Note that the introduction of underscores to a plugin filename does not prevent you from having commands such as `kubectl foo_bar`. The command from the above example, can be invoked using either a dash (-) or an underscore (_):

```
# You can invoke your custom command with a dash
kubectl foo-bar
```

```
I am a plugin with a dash in my name
```

```
# You can also invoke your custom command with an underscore  
kubectl foo_bar
```

```
I am a plugin with a dash in my name
```

Name conflicts and overshadowing

It is possible to have multiple plugins with the same filename in different locations throughout your `PATH`. For example, given a `PATH` with the following value: `PATH=/usr/local/bin/plugins:/usr/local/bin/moreplugins`, a copy of plugin `kubectl-foo` could exist in `/usr/local/bin/plugins` and `/usr/local/bin/moreplugins`, such that the output of the `kubectl` plugin list command is:

```
PATH=/usr/local/bin/plugins:/usr/local/bin/moreplugins kubectl plugin
```

```
The following kubectl-compatible plugins are available:
```

```
/usr/local/bin/plugins/kubectl-foo  
/usr/local/bin/moreplugins/kubectl-foo  
  - warning: /usr/local/bin/moreplugins/kubectl-foo is overshadowed by  
  
error: one plugin warning was found
```

In the above scenario, the warning under

`/usr/local/bin/moreplugins/kubectl-foo` tells you that this plugin will never be executed. Instead, the executable that appears first in your `PATH`, `/usr/local/bin/plugins/kubectl-foo`, will always be found and executed first by the `kubectl` plugin mechanism.

A way to resolve this issue is to ensure that the location of the plugin that you wish to use with `kubectl` always comes first in your `PATH`. For example, if you want to always use `/usr/local/bin/moreplugins/kubectl-foo` anytime that the `kubectl` command `kubectl foo` was invoked, change the value of your `PATH` to be

```
/usr/local/bin/moreplugins:/usr/local/bin/plugins .
```

Invocation of the longest executable filename

There is another kind of overshadowing that can occur with plugin filenames. Given two plugins present in a user's `PATH`: `kubectl-foo-bar` and `kubectl-foo-bar-baz`, the `kubectl` plugin mechanism will always choose the longest possible plugin name for a given user command. Some examples below, clarify this further:

```
# for a given kubectl command, the plugin with the longest possible fi  
kubectl foo bar baz
```

```
Plugin kubectl-foo-bar-baz is executed
```

```
kubectl foo bar
```

```
Plugin kubectl-foo-bar is executed
```

```
kubectl foo bar baz buz
```

```
Plugin kubectl-foo-bar-baz is executed, with "buz" as its first argument
```

```
kubectl foo bar buz
```

```
Plugin kubectl-foo-bar is executed, with "buz" as its first argument
```

This design choice ensures that plugin sub-commands can be implemented across multiple files, if needed, and that these sub-commands can be nested under a "parent" plugin command:

```
ls ./plugin_command_tree
```

```
kubectl-parent
kubectl-parent-subcommand
kubectl-parent-subcommand-subsubcommand
```

Checking for plugin warnings

You can use the aforementioned `kubectl plugin list` command to ensure that your plugin is visible by `kubectl`, and verify that there are no warnings preventing it from being called as a `kubectl` command.

```
kubectl plugin list
```

```
The following kubectl-compatible plugins are available:
```

```
test/fixtures/pkg/kubectl/plugins/kubectl-foo
/usr/local/bin/kubectl-foo
  - warning: /usr/local/bin/kubectl-foo is overshadowed by a similarly
    named plugin
  - warning: plugins/kubectl-invalid identified as a kubectl plugin, bu
error: 2 plugin warnings were found
```

Using the command line runtime package

If you're writing a plugin for kubectl and you're using Go, you can make use of the [cli-runtime](#) utility libraries.

These libraries provide helpers for parsing or updating a user's [kubeconfig](#) file, for making REST-style requests to the API server, or to bind flags associated with configuration and printing.

See the [Sample CLI Plugin](#) for an example usage of the tools provided in the CLI Runtime repo.

Distributing kubectl plugins

If you have developed a plugin for others to use, you should consider how you package it, distribute it and deliver updates to your users.

Krew

[Krew](#) offers a cross-platform way to package and distribute your plugins. This way, you use a single packaging format for all target platforms (Linux, Windows, macOS etc) and deliver updates to your users. Krew also maintains a [plugin index](#) so that other people can discover your plugin and install it.

Native / platform specific package management

Alternatively, you can use traditional package managers such as, `apt` or `yum` on Linux, Chocolatey on Windows, and Homebrew on macOS. Any package manager will be suitable if it can place new executables placed somewhere in the user's `PATH`. As a plugin author, if you pick this option then you also have the burden of updating your kubectl plugin's distribution package across multiple platforms for each release.

Source code

You can publish the source code; for example, as a Git repository. If you choose this option, someone who wants to use that plugin must fetch the code, set up a build environment (if it needs compiling), and deploy the plugin. If you also make compiled packages available, or use Krew, that will make installs easier.

What's next

- Check the Sample CLI Plugin repository for a [detailed example](#) of a plugin written in Go. In case of any questions, feel free to reach out to the [SIG CLI team](#).
- Read about [Krew](#), a package manager for kubectl plugins.

16 - Manage HugePages

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

FEATURE STATE: [Kubernetes v1.28 \[stable\]](#)

Kubernetes supports the allocation and consumption of pre-allocated huge pages by applications in a Pod. This page describes how users can consume huge pages.

Before you begin

Kubernetes nodes must [pre-allocate huge pages](#) in order for the node to report its huge page capacity.

A node can pre-allocate huge pages for multiple sizes, for instance, the following line in `/etc/default/grub` allocates `2*1GiB` of 1 GiB and `512*2 MiB` of 2 MiB pages:

```
GRUB_CMDLINE_LINUX="hugepagesz=1G hugepages=2 hugepagesz=2M hugepages=512 hugepages=1Gi hugepages=2Mi"
```

The nodes will automatically discover and report all huge page resources as schedulable resources.

When you describe the Node, you should see something similar to the following in the following in the `capacity` and `Allocatable` sections:

```
Capacity:  
cpu: ...  
ephemeral-storage: ...  
hugepages-1Gi: 2Gi  
hugepages-2Mi: 1Gi  
memory: ...  
pods: ...  
Allocatable:  
cpu: ...  
ephemeral-storage: ...  
hugepages-1Gi: 2Gi  
hugepages-2Mi: 1Gi  
memory: ...  
pods: ...
```

Note: For dynamically allocated pages (after boot), the Kubelet needs to be restarted for the new allocations to be rereflected.

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 and 1048576KiB page sizes, it will expose a schedulable resources `hugepages-2Mi` and `hugepages-1Gi`. Unlike CPU or memory, huge pages do not support overcommit. Note that when requesting hugepage resources, either memory or CPU resources must be requested as well.

A pod may consume multiple huge page sizes in a single pod spec. In this case it must use `medium: HugePages-<hugepagesize>` notation for all volume mounts.

```
apiVersion: v1
kind: Pod
metadata:
  name: huge-pages-example
spec:
  containers:
    - name: example
      image: fedora:latest
      command:
        - sleep
        - inf
      volumeMounts:
        - mountPath: /hugepages-2Mi
          name: hugepage-2mi
        - mountPath: /hugepages-1Gi
          name: hugepage-1gi
      resources:
        limits:
          hugepages-2Mi: 100Mi
          hugepages-1Gi: 2Gi
          memory: 100Mi
        requests:
          memory: 100Mi
      volumes:
        - name: hugepage-2mi
          emptyDir:
            medium: HugePages-2Mi
        - name: hugepage-1gi
          emptyDir:
            medium: HugePages-1Gi
```

A pod may use `medium: HugePages` only if it requests huge pages of one size.

```
apiVersion: v1
kind: Pod
metadata:
  name: huge-pages-example
spec:
  containers:
    - name: example
      image: fedora:latest
      command:
        - sleep
        - inf
      volumeMounts:
        - mountPath: /hugepages
          name: hugepage
      resources:
        limits:
          hugepages-2Mi: 100Mi
          memory: 100Mi
        requests:
          memory: 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 container scope, so each container has own limit on their cgroup sandbox as requested in a container spec.
- 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.

17 - Schedule GPUs

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

FEATURE STATE: [Kubernetes v1.26 \[stable\]](#)

Kubernetes includes **stable** support for managing AMD and NVIDIA GPUs (graphical processing units) across different nodes in your cluster, using device plugins.

This page describes how users can consume GPUs, and outlines some of the limitations in the implementation.

Using device plugins

Kubernetes implements device plugins to let Pods access specialized hardware features such as GPUs.

Note: This section links to third party projects that provide functionality required by Kubernetes. The Kubernetes project authors aren't responsible for these projects, which are listed alphabetically. To add a project to this list, read the [content guide](#) before submitting a change. [More information.](#)

As an administrator, you have to install GPU drivers from the corresponding hardware vendor on the nodes and run the corresponding device plugin from the GPU vendor. Here are some links to vendors' instructions:

- [AMD](#)
- [Intel](#)
- [NVIDIA](#)

Once you have installed the plugin, your cluster exposes a custom schedulable resource such as `amd.com/gpu` or `nvidia.com/gpu`.

You can consume these GPUs from your containers by requesting the custom GPU resource, the same way you request `cpu` or `memory`. However, there are some limitations in how you specify the resource requirements for custom devices.

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`.

Here's an example manifest for a Pod that requests a GPU:

```
apiVersion: v1
kind: Pod
metadata:
  name: example-vector-add
spec:
  restartPolicy: OnFailure
  containers:
    - name: example-vector-add
      image: "registry.example/example-vector-add:v42"
      resources:
        limits:
          gpu-vendor.example/example-gpu: 1 # requesting 1 GPU
```

Clusters containing different types of GPUs

If different nodes in your cluster have different types of 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 node1 accelerator=example-gpu-x100
kubectl label nodes node2 accelerator=other-gpu-k915
```

That label key `accelerator` is just an example; you can use a different label key if you prefer.

Automatic node labelling

If you're using AMD GPU devices, you can deploy [Node Labeller](#). Node Labeller is a [controller](#) that automatically labels your nodes with GPU device properties.

Similar functionality for NVIDIA is provided by [GPU feature discovery](#).