

Concepts

The Concepts section helps you learn about the parts of the Kubernetes system and the abstractions Kubernetes uses to represent your [cluster](#), and helps you obtain a deeper understanding of how Kubernetes works.

[Overview](#)

Get a high-level outline of Kubernetes and the components it is built from.

[Cluster Architecture](#)

The architectural concepts behind Kubernetes.

[Containers](#)

Technology for packaging an application along with its runtime dependencies.

[Workloads](#)

Understand Pods, the smallest deployable compute object in Kubernetes, and the higher-level abstractions that help you to run them.

[Services, Load Balancing, and Networking](#)

Concepts and resources behind networking in Kubernetes.

[Storage](#)

Ways to provide both long-term and temporary storage to Pods in your cluster.

[Configuration](#)

Resources that Kubernetes provides for configuring Pods.

[Security](#)

Concepts for keeping your cloud-native workload secure.

[Policies](#)

Policies you can configure that apply to groups of resources.

[Scheduling and Eviction](#)

In Kubernetes, scheduling refers to making sure that Pods are matched to Nodes so that the kubelet can run them. Eviction is the process of proactively failing one or more Pods on resource-starved Nodes.

[Cluster Administration](#)

Lower-level detail relevant to creating or administering a Kubernetes cluster.

[Extending Kubernetes](#)

Different ways to change the behavior of your Kubernetes cluster.

Overview

Get a high-level outline of Kubernetes and the components it is built from.

[What is Kubernetes?](#)

Kubernetes is a portable, extensible, open-source platform for managing containerized workloads and services, that facilitates both declarative configuration and automation. It has a large, rapidly growing ecosystem. Kubernetes services, support, and tools are widely available.

[Kubernetes Components](#)

A Kubernetes cluster consists of the components that represent the control plane and a set of machines called nodes.

[The Kubernetes API](#)

The Kubernetes API lets you query and manipulate the state of objects in Kubernetes. The core of Kubernetes' control plane is the API server and the HTTP API that it exposes. Users, the different parts of your cluster, and external components all communicate with one another through the API server.

[Working with Kubernetes Objects](#)

Kubernetes objects are persistent entities in the Kubernetes system. Kubernetes uses these entities to represent the state of your cluster. Learn about the Kubernetes object model and how to work with these objects.

What is Kubernetes?

Kubernetes is a portable, extensible, open-source platform for managing containerized workloads and services, that facilitates both declarative configuration and automation. It has a large, rapidly growing ecosystem. Kubernetes services, support, and tools are widely available.

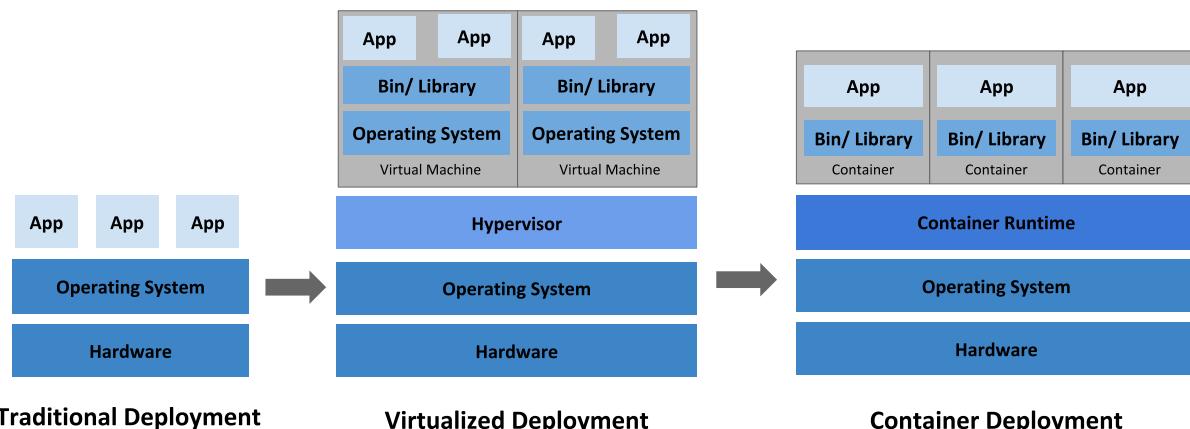
This page is an overview of Kubernetes.

Kubernetes is a portable, extensible, open-source platform for managing containerized workloads and services, that facilitates both declarative configuration and automation. It has a large, rapidly growing ecosystem. Kubernetes services, support, and tools are widely available.

The name Kubernetes originates from Greek, meaning helmsman or pilot. Google open-sourced the Kubernetes project in 2014. Kubernetes combines [over 15 years of Google's experience](#) running production workloads at scale with best-of-breed ideas and practices from the community.

Going back in time

Let's take a look at why Kubernetes is so useful by going back in time.



Traditional deployment era: Early on, organizations ran applications on physical servers. There was no way to define resource boundaries for applications in a physical server, and this caused resource allocation issues. For example, if multiple applications run on a physical server, there can be instances where one application would take up most of the resources, and as a result, the other applications would underperform. A solution for this would be to run each application on a different physical server. But this did not scale as resources were underutilized, and it was expensive for organizations to maintain many physical servers.

Virtualized deployment era: As a solution, virtualization was introduced. It allows you to run multiple Virtual Machines (VMs) on a single physical server's CPU. Virtualization allows applications to be isolated between VMs

and provides a level of security as the information of one application cannot be freely accessed by another application.

Virtualization allows better utilization of resources in a physical server and allows better scalability because an application can be added or updated easily, reduces hardware costs, and much more. With virtualization you can present a set of physical resources as a cluster of disposable virtual machines.

Each VM is a full machine running all the components, including its own operating system, on top of the virtualized hardware.

Container deployment era: Containers are similar to VMs, but they have relaxed isolation properties to share the Operating System (OS) among the applications. Therefore, containers are considered lightweight. Similar to a VM, a container has its own filesystem, share of CPU, memory, process space, and more. As they are decoupled from the underlying infrastructure, they are portable across clouds and OS distributions.

Containers have become popular because they provide extra benefits, such as:

- Agile application creation and deployment: increased ease and efficiency of container image creation compared to VM image use.
- Continuous development, integration, and deployment: provides for reliable and frequent container image build and deployment with quick and easy rollbacks (due to image immutability).
- Dev and Ops separation of concerns: create application container images at build/release time rather than deployment time, thereby decoupling applications from infrastructure.
- Observability not only surfaces OS-level information and metrics, but also application health and other signals.
- Environmental consistency across development, testing, and production: Runs the same on a laptop as it does in the cloud.
- Cloud and OS distribution portability: Runs on Ubuntu, RHEL, CoreOS, on-premises, on major public clouds, and anywhere else.
- Application-centric management: Raises the level of abstraction from running an OS on virtual hardware to running an application on an OS using logical resources.
- Loosely coupled, distributed, elastic, liberated micro-services: applications are broken into smaller, independent pieces and can be deployed and managed dynamically - not a monolithic stack running on one big single-purpose machine.
- Resource isolation: predictable application performance.
- Resource utilization: high efficiency and density.

Why you need Kubernetes and what it can do

Containers are a good way to bundle and run your applications. In a production environment, you need to manage the containers that run the applications and ensure that there is no downtime. For example, if a

container goes down, another container needs to start. Wouldn't it be easier if this behavior was handled by a system?

That's how Kubernetes comes to the rescue! Kubernetes provides you with a framework to run distributed systems resiliently. It takes care of scaling and failover for your application, provides deployment patterns, and more. For example, Kubernetes can easily manage a canary deployment for your system.

Kubernetes provides you with:

- **Service discovery and load balancing** Kubernetes can expose a container using the DNS name or using their own IP address. If traffic to a container is high, Kubernetes is able to load balance and distribute the network traffic so that the deployment is stable.
- **Storage orchestration** Kubernetes allows you to automatically mount a storage system of your choice, such as local storages, public cloud providers, and more.
- **Automated rollouts and rollbacks** You can describe the desired state for your deployed containers using Kubernetes, and it can change the actual state to the desired state at a controlled rate. For example, you can automate Kubernetes to create new containers for your deployment, remove existing containers and adopt all their resources to the new container.
- **Automatic bin packing** You provide Kubernetes with a cluster of nodes that it can use to run containerized tasks. You tell Kubernetes how much CPU and memory (RAM) each container needs. Kubernetes can fit containers onto your nodes to make the best use of your resources.
- **Self-healing** Kubernetes restarts containers that fail, replaces containers, kills containers that don't respond to your user-defined health check, and doesn't advertise them to clients until they are ready to serve.
- **Secret and configuration management** Kubernetes lets you store and manage sensitive information, such as passwords, OAuth tokens, and SSH keys. You can deploy and update secrets and application configuration without rebuilding your container images, and without exposing secrets in your stack configuration.

What Kubernetes is not

Kubernetes is not a traditional, all-inclusive PaaS (Platform as a Service) system. Since Kubernetes operates at the container level rather than at the hardware level, it provides some generally applicable features common to PaaS offerings, such as deployment, scaling, load balancing, and lets users integrate their logging, monitoring, and alerting solutions. However, Kubernetes is not monolithic, and these default solutions are optional and pluggable. Kubernetes provides the building blocks for building developer platforms, but preserves user choice and flexibility where it is important.

Kubernetes:

- Does not limit the types of applications supported. Kubernetes aims to support an extremely diverse variety of workloads, including stateless, stateful, and data-processing workloads. If an application can run in a container, it should run great on Kubernetes.
- Does not deploy source code and does not build your application. Continuous Integration, Delivery, and Deployment (CI/CD) workflows are determined by organization cultures and preferences as well as technical requirements.
- Does not provide application-level services, such as middleware (for example, message buses), data-processing frameworks (for example, Spark), databases (for example, MySQL), caches, nor cluster storage systems (for example, Ceph) as built-in services. Such components can run on Kubernetes, and/or can be accessed by applications running on Kubernetes through portable mechanisms, such as the [Open Service Broker](#).
- Does not dictate logging, monitoring, or alerting solutions. It provides some integrations as proof of concept, and mechanisms to collect and export metrics.
- Does not provide nor mandate a configuration language/system (for example, Jsonnet). It provides a declarative API that may be targeted by arbitrary forms of declarative specifications.
- Does not provide nor adopt any comprehensive machine configuration, maintenance, management, or self-healing systems.
- Additionally, Kubernetes is not a mere orchestration system. In fact, it eliminates the need for orchestration. The technical definition of orchestration is execution of a defined workflow: first do A, then B, then C. In contrast, Kubernetes comprises a set of independent, composable control processes that continuously drive the current state towards the provided desired state. It shouldn't matter how you get from A to C. Centralized control is also not required. This results in a system that is easier to use and more powerful, robust, resilient, and extensible.

What's next

- Take a look at the [Kubernetes Components](#)
- Ready to [Get Started?](#)

Feedback

Was this page helpful?

Yes No

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

Last modified October 22, 2020 at 6:45 AM PST: [Containers don't get their own CPU \(#24659\) \(fcdfba35\)](#)

- [Going back in time](#)
- [Why you need Kubernetes and what it can do](#)
- [What Kubernetes is not](#)
- [What's next](#)

Kubernetes Components

A Kubernetes cluster consists of the components that represent the control plane and a set of machines called nodes.

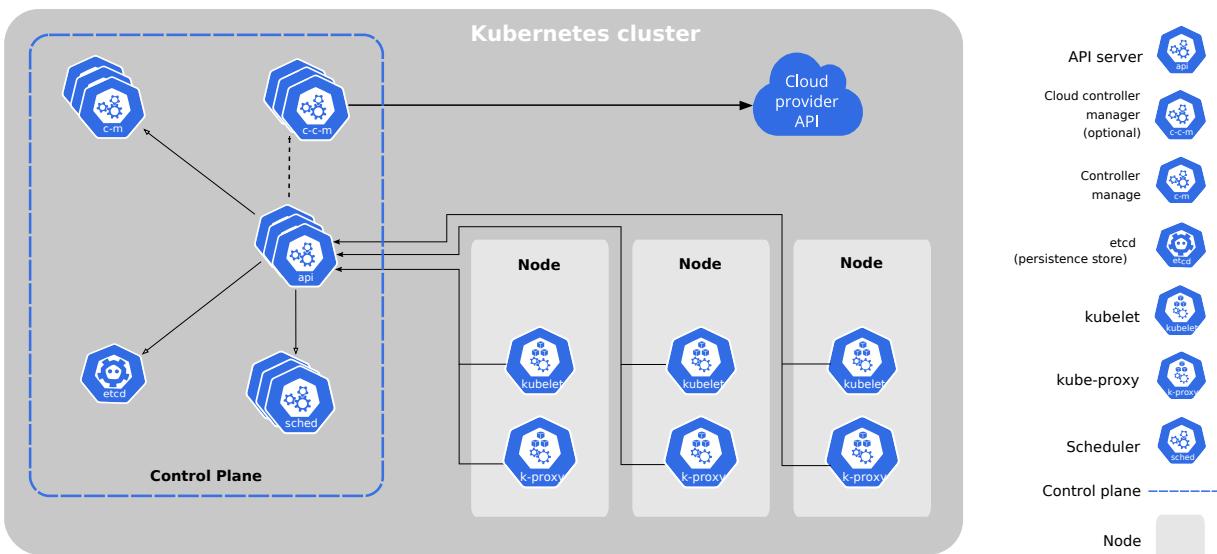
When you deploy Kubernetes, you get a cluster.

A Kubernetes cluster consists of a set of worker machines, called [nodes](#), that run containerized applications. Every cluster has at least one worker node.

The worker node(s) host the [Pods](#) that are the components of the application workload. The [control plane](#) manages the worker nodes and the Pods in the cluster. In production environments, the control plane usually runs across multiple computers and a cluster usually runs multiple nodes, providing fault-tolerance and high availability.

This document outlines the various components you need to have a complete and working Kubernetes cluster.

Here's the diagram of a Kubernetes cluster with all the components tied together.



Control Plane Components

The control plane's components make global decisions about the cluster (for example, scheduling), as well as detecting and responding to cluster events

(for example, starting up a new [pod](#) when a deployment's `replicas` field is unsatisfied).

Control plane components can be run on any machine in the cluster. However, for simplicity, set up scripts typically start all control plane components on the same machine, and do not run user containers on this machine. See [Building High-Availability Clusters](#) for an example multi-master-VM setup.

kube-apiserver

The API server is a component of the Kubernetes [control plane](#) that exposes the Kubernetes API. The API server is the front end for the Kubernetes control plane.

The main implementation of a Kubernetes API server is [kube-apiserver](#). kube-apiserver is designed to scale horizontally—that is, it scales by deploying more instances. You can run several instances of kube-apiserver and balance traffic between those instances.

etcd

Consistent and highly-available key value store used as Kubernetes' backing store for all cluster data.

If your Kubernetes cluster uses etcd as its backing store, make sure you have a [back up](#) plan for those data.

You can find in-depth information about etcd in the official [documentation](#).

kube-scheduler

Control plane component that watches for newly created [Pods](#) with no assigned [node](#), and selects a node for them to run on.

Factors taken into account for scheduling decisions include: individual and collective resource requirements, hardware/software/policy constraints, affinity and anti-affinity specifications, data locality, inter-workload interference, and deadlines.

kube-controller-manager

Control Plane component that runs [controller](#) processes.

Logically, each [controller](#) is a separate process, but to reduce complexity, they are all compiled into a single binary and run in a single process.

These controllers include:

- *Node controller: Responsible for noticing and responding when nodes go down.*

- *Replication controller: Responsible for maintaining the correct number of pods for every replication controller object in the system.*
- *Endpoints controller: Populates the Endpoints object (that is, joins Services & Pods).*
- *Service Account & Token controllers: Create default accounts and API access tokens for new namespaces.*

cloud-controller-manager

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 just interact with your cluster.

The cloud-controller-manager only runs controllers that are specific to your cloud provider. If you are running Kubernetes on your own premises, or in a learning environment inside your own PC, the cluster does not have a cloud controller manager.

As with the kube-controller-manager, the cloud-controller-manager combines several logically independent control loops into a single binary that you run as a single process. You can scale horizontally (run more than one copy) to improve performance or to help tolerate failures.

The following controllers can have cloud provider dependencies:

- *Node controller: For checking the cloud provider to determine if a node has been deleted in the cloud after it stops responding*
- *Route controller: For setting up routes in the underlying cloud infrastructure*
- *Service controller: For creating, updating and deleting cloud provider load balancers*

Node Components

Node components run on every node, maintaining running pods and providing the Kubernetes runtime environment.

kubelet

An agent that runs on each [node](#) in the cluster. It makes sure that [containers](#) are running in a [Pod](#).

The kubelet takes a set of PodSpecs that are provided through various mechanisms and ensures that the containers described in those PodSpecs

are running and healthy. The kubelet doesn't manage containers which were not created by Kubernetes.

kube-proxy

kube-proxy is a network proxy that runs on each [node](#) in your cluster, implementing part of the Kubernetes [Service](#) concept.

[kube-proxy](#) maintains network rules on nodes. These network rules allow network communication to your Pods from network sessions inside or outside of your cluster.

kube-proxy uses the operating system packet filtering layer if there is one and it's available. Otherwise, kube-proxy forwards the traffic itself.

Container runtime

The container runtime is the software that is responsible for running containers.

Kubernetes supports several container runtimes: [Docker](#), [containerd](#), [CRI-O](#), and any implementation of the [Kubernetes CRI \(Container Runtime Interface\)](#).

Addons

Addons use Kubernetes resources ([DaemonSet](#), [Deployment](#), etc) to implement cluster features. Because these are providing cluster-level features, namespaced resources for addons belong within the `kube-system` namespace.

Selected addons are described below; for an extended list of available addons, please see [Addons](#).

DNS

While the other addons are not strictly required, all Kubernetes clusters should have [cluster DNS](#), as many examples rely on it.

Cluster DNS is a DNS server, in addition to the other DNS server(s) in your environment, which serves DNS records for Kubernetes services.

Containers started by Kubernetes automatically include this DNS server in their DNS searches.

Web UI (Dashboard)

[Dashboard](#) is a general purpose, web-based UI for Kubernetes clusters. It allows users to manage and troubleshoot applications running in the cluster, as well as the cluster itself.

Container Resource Monitoring

[Container Resource Monitoring](#) records generic time-series metrics about containers in a central database, and provides a UI for browsing that data.

Cluster-level Logging

A [cluster-level logging](#) mechanism is responsible for saving container logs to a central log store with search/browsing interface.

What's next

- Learn about [Nodes](#)
- Learn about [Controllers](#)
- Learn about [kube-scheduler](#)
- Read etcd's official [documentation](#)

Feedback

Was this page helpful?

Yes No

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

Last modified August 28, 2020 at 4:49 PM PST: [Update Kubernetes architectural diagram \(#23138\) \(94b2f857b\)](#)

- [Control Plane Components](#)
 - [kube-apiserver](#)
 - [etcd](#)
 - [kube-scheduler](#)
 - [kube-controller-manager](#)
 - [cloud-controller-manager](#)
- [Node Components](#)
 - [kubelet](#)
 - [kube-proxy](#)
 - [Container runtime](#)
- [Addons](#)
 - [DNS](#)
 - [Web UI \(Dashboard\)](#)
 - [Container Resource Monitoring](#)
 - [Cluster-level Logging](#)
- [What's next](#)

The Kubernetes API

The Kubernetes API lets you query and manipulate the state of objects in Kubernetes. The core of Kubernetes' control plane is the API server and the HTTP API that it exposes. Users, the different parts of your cluster, and external components all communicate with one another through the API server.

The core of Kubernetes' [control plane](#) is the [API server](#). The API server exposes an HTTP API that lets end users, different parts of your cluster, and external components communicate with one another.

The Kubernetes API lets you query and manipulate the state of API objects in Kubernetes (for example: Pods, Namespaces, ConfigMaps, and Events).

Most operations can be performed through the [kubectl](#) command-line interface or other command-line tools, such as [kubeadm](#), which in turn use the API. However, you can also access the API directly using REST calls.

Consider using one of the [client libraries](#) if you are writing an application using the Kubernetes API.

OpenAPI specification

Complete API details are documented using [OpenAPI](#).

The Kubernetes API server serves an OpenAPI spec via the /openapi/v2 endpoint. You can request the response format using request headers as follows:

Header	Possible values	Notes
Accept-Encoding	gzip	<i>not supplying this header is also acceptable</i>
Accept	application/com.github.proto-openapi.spec.v2@v1.0+protobuf	<i>mainly for intra-cluster use</i>
	application/json	<i>default</i>
	*	<i>serves application/json</i>

Kubernetes implements an alternative Protobuf based serialization format that is primarily intended for intra-cluster communication. For more information about this format, see the [Kubernetes Protobuf serialization design proposal](#) and the Interface Definition Language (IDL) files for each schema located in the Go packages that define the API objects.

Persistence

Kubernetes stores the serialized state of objects by writing them into [etcd](#).

API groups and versioning

To make it easier to eliminate fields or restructure resource representations, Kubernetes supports multiple API versions, each at a different API path, such as /api/v1 or /apis/rbac.authorization.k8s.io/v1alpha1.

Versioning is done at the API level rather than at the resource or field level to ensure that the API presents a clear, consistent view of system resources and behavior, and to enable controlling access to end-of-life and/or experimental APIs.

To make it easier to evolve and to extend its API, Kubernetes implements [API groups](#) that can be [enabled or disabled](#).

API resources are distinguished by their API group, resource type, namespace (for namespaced resources), and name. The API server handles the conversion between API versions transparently: all the different versions are actually representations of the same persisted data. The API server may serve the same underlying data through multiple API versions.

For example, suppose there are two API versions, v1 and v1beta1, for the same resource. If you originally created an object using the v1beta1 version of its API, you can later read, update, or delete that object using either the v1beta1 or the v1 API version.

API changes

Any system that is successful needs to grow and change as new use cases emerge or existing ones change. Therefore, Kubernetes has designed the Kubernetes API to continuously change and grow. The Kubernetes project aims to not break compatibility with existing clients, and to maintain that

compatibility for a length of time so that other projects have an opportunity to adapt.

In general, new API resources and new resource fields can be added often and frequently. Elimination of resources or fields requires following the [API deprecation policy](#).

Kubernetes makes a strong commitment to maintain compatibility for official Kubernetes APIs once they reach general availability (GA), typically at API version v1. Additionally, Kubernetes keeps compatibility even for beta API versions wherever feasible: if you adopt a beta API you can continue to interact with your cluster using that API, even after the feature goes stable.

Note: *Although Kubernetes also aims to maintain compatibility for alpha APIs versions, in some circumstances this is not possible. If you use any alpha API versions, check the release notes for Kubernetes when upgrading your cluster, in case the API did change.*

Refer to [API versions reference](#) for more details on the API version level definitions.

API Extension

The Kubernetes API can be extended in one of two ways:

1. [Custom resources](#) let you declaratively define how the API server should provide your chosen resource API.
2. You can also extend the Kubernetes API by implementing an [aggregation layer](#).

What's next

- Learn how to extend the Kubernetes API by adding your own [CustomResourceDefinition](#).
- [Controlling Access To The Kubernetes API](#) describes how the cluster manages authentication and authorization for API access.
- Learn about API endpoints, resource types and samples by reading [API Reference](#).
- Learn about what constitutes a compatible change, and how to change the API, from [API changes](#).

Feedback

Was this page helpful?

Yes No

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

Last modified December 08, 2020 at 1:07 PM PST: [Make wording in overview more in line with the description \(9c3891144\)](#)
[Edit this page](#) [Create child page](#) [Create an issue](#)

- [OpenAPI specification](#)
- [Persistence](#)
- [API groups and versioning](#)
 - [API changes](#)
- [API Extension](#)
- [What's next](#)

Working with Kubernetes Objects

Kubernetes objects are persistent entities in the Kubernetes system. Kubernetes uses these entities to represent the state of your cluster. Learn about the Kubernetes object model and how to work with these objects.

[**Understanding Kubernetes Objects**](#)

[**Kubernetes Object Management**](#)

[**Object Names and IDs**](#)

[**Namespaces**](#)

[**Labels and Selectors**](#)

[**Annotations**](#)

[**Field Selectors**](#)

[**Recommended Labels**](#)

Understanding Kubernetes Objects

This page explains how Kubernetes objects are represented in the Kubernetes API, and how you can express them in `.yaml` format.

Understanding Kubernetes objects

Kubernetes objects are persistent entities in the Kubernetes system. Kubernetes uses these entities to represent the state of your cluster. Specifically, they can describe:

- What containerized applications are running (and on which nodes)
- The resources available to those applications

- The policies around how those applications behave, such as restart policies, upgrades, and fault-tolerance

A Kubernetes object is a "record of intent"--once you create the object, the Kubernetes system will constantly work to ensure that object exists. By creating an object, you're effectively telling the Kubernetes system what you want your cluster's workload to look like; this is your cluster's desired state.

To work with Kubernetes objects--whether to create, modify, or delete them--you'll need to use the [Kubernetes API](#). When you use the `kubectl` command-line interface, for example, the CLI makes the necessary Kubernetes API calls for you. You can also use the Kubernetes API directly in your own programs using one of the [Client Libraries](#).

Object Spec and Status

Almost every Kubernetes object includes two nested object fields that govern the object's configuration: the *object spec* and the *object status*. For objects that have a *spec*, you have to set this when you create the object, providing a description of the characteristics you want the resource to have: its desired state.

The *status* describes the current state of the object, supplied and updated by the Kubernetes system and its components. The Kubernetes [control plane](#) continually and actively manages every object's actual state to match the desired state you supplied.

For example: in Kubernetes, a *Deployment* is an object that can represent an application running on your cluster. When you create the *Deployment*, you might set the *Deployment spec* to specify that you want three replicas of the application to be running. The Kubernetes system reads the *Deployment spec* and starts three instances of your desired application--updating the *status* to match your *spec*. If any of those instances should fail (a *status change*), the Kubernetes system responds to the difference between *spec* and *status* by making a correction--in this case, starting a replacement instance.

For more information on the *object spec*, *status*, and *metadata*, see the [Kubernetes API Conventions](#).

Describing a Kubernetes object

When you create an object in Kubernetes, you must provide the *object spec* that describes its desired state, as well as some basic information about the object (such as a name). When you use the Kubernetes API to create the object (either directly or via `kubectl`), that API request must include that information as JSON in the request body. **Most often, you provide the information to `kubectl` in a .yaml file.** `kubectl` converts the information to JSON when making the API request.

Here's an example `.yaml` file that shows the required fields and *object spec* for a Kubernetes *Deployment*:



```
apiVersion: apps/v1 # for versions before 1.9.0 use apps/v1beta2
kind: Deployment
metadata:
  name: nginx-deployment
spec:
  selector:
    matchLabels:
      app: nginx
  replicas: 2 # tells deployment to run 2 pods matching the template
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
        - name: nginx
          image: nginx:1.14.2
          ports:
            - containerPort: 80
```

One way to create a Deployment using a `.yaml` file like the one above is to use the [`kubectl apply`](#) command in the `kubectl` command-line interface, passing the `.yaml` file as an argument. Here's an example:

```
kubectl apply -f https://k8s.io/examples/application/
deployment.yaml --record
```

The output is similar to this:

```
deployment.apps/nginx-deployment created
```

Required Fields

In the `.yaml` file for the Kubernetes object you want to create, you'll need to set values for the following fields:

- `apiVersion` - Which version of the Kubernetes API you're using to create this object
- `kind` - What kind of object you want to create
- `metadata` - Data that helps uniquely identify the object, including a name string, UID, and optional namespace
- `spec` - What state you desire for the object

The precise format of the object `spec` is different for every Kubernetes object, and contains nested fields specific to that object. The [Kubernetes API Reference](#) can help you find the `spec` format for all of the objects you can create using Kubernetes. For example, the `spec` format for a Pod can be

found in [PodSpec v1 core](#), and the `spec` format for a Deployment can be found in [DeploymentSpec v1 apps](#).

What's next

- Learn about the most important basic Kubernetes objects, such as [Pod](#).
- Learn about [controllers](#) in Kubernetes.
- [Using the Kubernetes API](#) explains some more API concepts.

Feedback

Was this page helpful?

Yes No

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

Last modified October 13, 2020 at 12:41 AM PST: [Move API overview to be a Docsy section overview \(3edb97057\)](#)
[Edit this page](#) [Create child page](#) [Create an issue](#)

- [Understanding Kubernetes objects](#)
 - [Object Spec and Status](#)
 - [Describing a Kubernetes object](#)
 - [Required Fields](#)
- [What's next](#)

Kubernetes Object Management

The `kubectl` command-line tool supports several different ways to create and manage Kubernetes objects. This document provides an overview of the different approaches. Read the [Kubectl book](#) for details of managing objects by Kubectl.

Management techniques

Warning: A Kubernetes object should be managed using only one technique. Mixing and matching techniques for the same object results in undefined behavior.

Management technique	Operates on	Recommended environment	Supported writers	Learning curve
Imperative commands	Live objects	Development projects	1+	Lowest
Imperative object configuration	Individual files	Production projects	1	Moderate

Management technique	Operates on	Recommended environment	Supported writers	Learning curve
Declarative object configuration	Directories of files	Production projects	1+	Highest

Imperative commands

When using imperative commands, a user operates directly on live objects in a cluster. The user provides operations to the `kubectl` command as arguments or flags.

This is the simplest way to get started or to run a one-off task in a cluster. Because this technique operates directly on live objects, it provides no history of previous configurations.

Examples

Run an instance of the `nginx` container by creating a `Deployment` object:

```
kubectl create deployment nginx --image nginx
```

Trade-offs

Advantages compared to object configuration:

- *Commands are simple, easy to learn and easy to remember.*
- *Commands require only a single step to make changes to the cluster.*

Disadvantages compared to object configuration:

- *Commands do not integrate with change review processes.*
- *Commands do not provide an audit trail associated with changes.*
- *Commands do not provide a source of records except for what is live.*
- *Commands do not provide a template for creating new objects.*

Imperative object configuration

In imperative object configuration, the `kubectl` command specifies the operation (create, replace, etc.), optional flags and at least one file name. The file specified must contain a full definition of the object in YAML or JSON format.

See the [API reference](#) for more details on object definitions.

Warning: The imperative `replace` command replaces the existing spec with the newly provided one, dropping all changes to the object missing from the configuration file. This approach should not be used with resource types whose specs are updated independently of the configuration file. Services of type `LoadBalancer`, for example, have their `externalIPs` field updated independently from the configuration by the cluster.

Examples

Create the objects defined in a configuration file:

```
kubectl create -f nginx.yaml
```

Delete the objects defined in two configuration files:

```
kubectl delete -f nginx.yaml -f redis.yaml
```

Update the objects defined in a configuration file by overwriting the live configuration:

```
kubectl replace -f nginx.yaml
```

Trade-offs

Advantages compared to imperative commands:

- *Object configuration can be stored in a source control system such as Git.*
- *Object configuration can integrate with processes such as reviewing changes before push and audit trails.*
- *Object configuration provides a template for creating new objects.*

Disadvantages compared to imperative commands:

- *Object configuration requires basic understanding of the object schema.*
- *Object configuration requires the additional step of writing a YAML file.*

Advantages compared to declarative object configuration:

- *Imperative object configuration behavior is simpler and easier to understand.*
- *As of Kubernetes version 1.5, imperative object configuration is more mature.*

Disadvantages compared to declarative object configuration:

- *Imperative object configuration works best on files, not directories.*
- *Updates to live objects must be reflected in configuration files, or they will be lost during the next replacement.*

Declarative object configuration

When using declarative object configuration, a user operates on object configuration files stored locally, however the user does not define the operations to be taken on the files. Create, update, and delete operations are automatically detected per-object by kubectl. This enables working on directories, where different operations might be needed for different objects.

Note: Declarative object configuration retains changes made by other writers, even if the changes are not merged back to the object configuration file. This is possible by using the patch API operation to write only observed differences, instead of using the replace API operation to replace the entire object configuration.

Examples

Process all object configuration files in the `configs` directory, and create or patch the live objects. You can first `diff` to see what changes are going to be made, and then apply:

```
kubectl diff -f configs/  
kubectl apply -f configs/
```

Recursively process directories:

```
kubectl diff -R -f configs/  
kubectl apply -R -f configs/
```

Trade-offs

Advantages compared to imperative object configuration:

- Changes made directly to live objects are retained, even if they are not merged back into the configuration files.
- Declarative object configuration has better support for operating on directories and automatically detecting operation types (create, patch, delete) per-object.

Disadvantages compared to imperative object configuration:

- Declarative object configuration is harder to debug and understand results when they are unexpected.
- Partial updates using `diffs` create complex merge and patch operations.

What's next

- [Managing Kubernetes Objects Using Imperative Commands](#)
- [Managing Kubernetes Objects Using Object Configuration \(Imperative\)](#)
- [Managing Kubernetes Objects Using Object Configuration \(Declarative\)](#)
- [Managing Kubernetes Objects Using Kustomize \(Declarative\)](#)
- [Kubectl Command Reference](#)
- [Kubectl Book](#)
- [Kubernetes API Reference](#)

Feedback

Was this page helpful?

Yes No

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

Last modified July 17, 2020 at 4:11 PM PST: [Fix links in concepts section \(2\)](#) ([c8f470487](#))

[Edit this page](#) [Create child page](#) [Create an issue](#)

- [Management techniques](#)
- [Imperative commands](#)
 - [Examples](#)
 - [Trade-offs](#)
- [Imperative object configuration](#)
 - [Examples](#)
 - [Trade-offs](#)
- [Declarative object configuration](#)
 - [Examples](#)
 - [Trade-offs](#)
- [What's next](#)

Object Names and IDs

Each object in your cluster has a [Name](#) that is unique for that type of resource. Every Kubernetes object also has a [UID](#) that is unique across your whole cluster.

For example, you can only have one Pod named `myapp-1234` within the same [namespace](#), but you can have one Pod and one Deployment that are each named `myapp-1234`.

For non-unique user-provided attributes, Kubernetes provides [labels](#) and [annotations](#).

Names

A client-provided string that refers to an object in a resource URL, such as `/api/v1/pods/some-name`.

Only one object of a given kind can have a given name at a time. However, if you delete the object, you can make a new object with the same name.

Below are three types of commonly used name constraints for resources.

DNS Subdomain Names

Most resource types require a name that can be used as a DNS subdomain name as defined in [RFC 1123](#). This means the name must:

- contain no more than 253 characters
- contain only lowercase alphanumeric characters, '-' or '.'
- start with an alphanumeric character

- end with an alphanumeric character

DNS Label Names

Some resource types require their names to follow the DNS label standard as defined in [RFC 1123](#). This means the name must:

- contain at most 63 characters
- contain only lowercase alphanumeric characters or '-'
- start with an alphanumeric character
- end with an alphanumeric character

Path Segment Names

Some resource types require their names to be able to be safely encoded as a path segment. In other words, the name may not be "." or ".." and the name may not contain "/" or "%".

Here's an example manifest for a Pod named nginx-demo.

```
apiVersion: v1
kind: Pod
metadata:
  name: nginx-demo
spec:
  containers:
  - name: nginx
    image: nginx:1.14.2
    ports:
    - containerPort: 80
```

Note: Some resource types have additional restrictions on their names.

UIDs

A Kubernetes systems-generated string to uniquely identify objects.

Every object created over the whole lifetime of a Kubernetes cluster has a distinct UID. It is intended to distinguish between historical occurrences of similar entities.

Kubernetes UIDs are universally unique identifiers (also known as UUIDs). UUIDs are standardized as ISO/IEC 9834-8 and as ITU-T X.667.

What's next

- Read about [labels](#) in Kubernetes.
- See the [Identifiers and Names in Kubernetes](#) design document.

Feedback

Was this page helpful?

Yes No

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

Last modified August 05, 2020 at 3:17 AM PST: [Replace special quote characters with normal ones. \(c6a96128c\)](#)

[Edit this page](#) [Create child page](#) [Create an issue](#)

- [Names](#)
 - [DNS Subdomain Names](#)
 - [DNS Label Names](#)
 - [Path Segment Names](#)
- [UIDs](#)
- [What's next](#)

Namespaces

Kubernetes supports multiple virtual clusters backed by the same physical cluster. These virtual clusters are called namespaces.

When to Use Multiple Namespaces

Namespaces are intended for use in environments with many users spread across multiple teams, or projects. For clusters with a few to tens of users, you should not need to create or think about namespaces at all. Start using namespaces when you need the features they provide.

Namespaces provide a scope for names. Names of resources need to be unique within a namespace, but not across namespaces. Namespaces cannot be nested inside one another and each Kubernetes resource can only be in one namespace.

Namespaces are a way to divide cluster resources between multiple users (via [resource quota](#)).

It is not necessary to use multiple namespaces just to separate slightly different resources, such as different versions of the same software: use [labels](#) to distinguish resources within the same namespace.

Working with Namespaces

Creation and deletion of namespaces are described in the [Admin Guide documentation for namespaces](#).

Note: Avoid creating namespace with prefix `kube-`, since it is reserved for Kubernetes system namespaces.

Viewing namespaces

You can list the current namespaces in a cluster using:

```
kubectl get namespace
```

NAME	STATUS	AGE
<code>default</code>	Active	1d
<code>kube-node-lease</code>	Active	1d
<code>kube-public</code>	Active	1d
<code>kube-system</code>	Active	1d

Kubernetes starts with four initial namespaces:

- `default` The default namespace for objects with no other namespace
- `kube-system` The namespace for objects created by the Kubernetes system
- `kube-public` This namespace is created automatically and is readable by all users (including those not authenticated). This namespace is mostly reserved for cluster usage, in case that some resources should be visible and readable publicly throughout the whole cluster. The public aspect of this namespace is only a convention, not a requirement.
- `kube-node-lease` This namespace for the lease objects associated with each node which improves the performance of the node heartbeats as the cluster scales.

Setting the namespace for a request

To set the namespace for a current request, use the `--namespace` flag.

For example:

```
kubectl run nginx --image=nginx --namespace=<insert-namespace-name-here>
kubectl get pods --namespace=<insert-namespace-name-here>
```

Setting the namespace preference

You can permanently save the namespace for all subsequent `kubectl` commands in that context.

```
kubectl config set-context --current --namespace=<insert-namespace-name-here>
# Validate it
kubectl config view --minify | grep namespace:
```

Namespaces and DNS

When you create a [Service](#), it creates a corresponding [DNS entry](#). This entry is of the form <service-name>. <namespace-name>. svc. cluster. local, which means that if a container just uses <service-name>, it will resolve to the service which is local to a namespace. This is useful for using the same configuration across multiple namespaces such as Development, Staging and Production. If you want to reach across namespaces, you need to use the fully qualified domain name (FQDN).

Not All Objects are in a Namespace

Most Kubernetes resources (e.g. pods, services, replication controllers, and others) are in some namespaces. However namespace resources are not themselves in a namespace. And low-level resources, such as [nodes](#) and [persistentVolumes](#), are not in any namespace.

To see which Kubernetes resources are and aren't in a namespace:

```
# In a namespace  
kubectl api-resources --namespaced=true  
  
# Not in a namespace  
kubectl api-resources --namespaced=false
```

What's next

- Learn more about [creating a new namespace](#).
- Learn more about [deleting a namespace](#).

Feedback

Was this page helpful?

Yes No

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

Last modified October 10, 2020 at 7:15 PM PST: [Remove statement about future of namespaces \(dd88014f3\)](#)

[Edit this page](#) [Create child page](#) [Create an issue](#)

- [When to Use Multiple Namespaces](#)
- [Working with Namespaces](#)
 - [Viewing namespaces](#)
 - [Setting the namespace for a request](#)
 - [Setting the namespace preference](#)
- [Namespaces and DNS](#)

- [Not All Objects are in a Namespace](#)
- [What's next](#)

Labels and Selectors

Labels are key/value pairs that are attached to objects, such as pods. Labels are intended to be used to specify identifying attributes of objects that are meaningful and relevant to users, but do not directly imply semantics to the core system. Labels can be used to organize and to select subsets of objects. Labels can be attached to objects at creation time and subsequently added and modified at any time. Each object can have a set of key/value labels defined. Each Key must be unique for a given object.

```
"metadata": {
  "labels": {
    "key1" : "value1",
    "key2" : "value2"
  }
}
```

Labels allow for efficient queries and watches and are ideal for use in UIs and CLIs. Non-identifying information should be recorded using [annotations](#).

Motivation

Labels enable users to map their own organizational structures onto system objects in a loosely coupled fashion, without requiring clients to store these mappings.

Service deployments and batch processing pipelines are often multi-dimensional entities (e.g., multiple partitions or deployments, multiple release tracks, multiple tiers, multiple micro-services per tier). Management often requires cross-cutting operations, which breaks encapsulation of strictly hierarchical representations, especially rigid hierarchies determined by the infrastructure rather than by users.

Example labels:

- "release" : "stable", "release" : "canary"
- "environment" : "dev", "environment" : "qa", "environment" : "production"
- "tier" : "frontend", "tier" : "backend", "tier" : "cache"
- "partition" : "customerA", "partition" : "customerB"
- "track" : "daily", "track" : "weekly"

These are just examples of commonly used labels; you are free to develop your own conventions. Keep in mind that label Key must be unique for a given object.

Syntax and character set

Labels are key/value pairs. Valid label keys have two segments: an optional prefix and name, separated by a slash (/). The name segment is required and must be 63 characters or less, beginning and ending with an alphanumeric character ([a-z0-9A-Z]) with dashes (-), underscores (_), dots (.), and alphanumerics between. The prefix is optional. If specified, the prefix must be a DNS subdomain: a series of DNS labels separated by dots (.), not longer than 253 characters in total, followed by a slash (/).

If the prefix is omitted, the label Key is presumed to be private to the user. Automated system components (e.g. kube-scheduler, kube-controller-manager, kube-apiserver, kubectl, or other third-party automation) which add labels to end-user objects must specify a prefix.

The kubernetes.io/ and k8s.io/ prefixes are reserved for Kubernetes core components.

Valid label values must be 63 characters or less and must be empty or begin and end with an alphanumeric character ([a-z0-9A-Z]) with dashes (-), underscores (_), dots (.), and alphanumerics between.

For example, here's the configuration file for a Pod that has two labels environment: production and app: nginx :

```
apiVersion: v1
kind: Pod
metadata:
  name: label-demo
  labels:
    environment: production
    app: nginx
spec:
  containers:
  - name: nginx
    image: nginx:1.14.2
    ports:
    - containerPort: 80
```

Label selectors

Unlike [names and UIDs](#), labels do not provide uniqueness. In general, we expect many objects to carry the same label(s).

Via a label selector, the client/user can identify a set of objects. The label selector is the core grouping primitive in Kubernetes.

The API currently supports two types of selectors: equality-based and set-based. A label selector can be made of multiple requirements which are

comma-separated. In the case of multiple requirements, all must be satisfied so the comma separator acts as a logical AND (&&) operator.

The semantics of empty or non-specified selectors are dependent on the context, and API types that use selectors should document the validity and meaning of them.

Note: For some API types, such as `ReplicaSets`, the label selectors of two instances must not overlap within a namespace, or the controller can see that as conflicting instructions and fail to determine how many replicas should be present.

Caution: For both equality-based and set-based conditions there is no logical OR (||) operator. Ensure your filter statements are structured accordingly.

Equality-based requirement

Equality- or inequality-based requirements allow filtering by label keys and values. Matching objects must satisfy all of the specified label constraints, though they may have additional labels as well. Three kinds of operators are admitted =, ==, !=. The first two represent equality (and are simply synonyms), while the latter represents inequality. For example:

```
environment = production
tier != frontend
```

The former selects all resources with key equal to `environment` and value equal to `production`. The latter selects all resources with key equal to `tier` and value distinct from `frontend`, and all resources with no labels with the `tier` key. One could filter for resources in `production` excluding `frontend` using the comma operator: `environment=production,tier!=frontend`

One usage scenario for equality-based label requirement is for Pods to specify node selection criteria. For example, the sample Pod below selects nodes with the label "accelerator=nvidia-tesla-p100".

```
apiVersion: v1
kind: Pod
metadata:
  name: cuda-test
spec:
  containers:
    - name: cuda-test
      image: "k8s.gcr.io/cuda-vector-add:v0.1"
      resources:
        limits:
          nvidia.com/gpu: 1
  nodeSelector:
    accelerator: nvidia-tesla-p100
```

Set-based requirement

Set-based label requirements allow filtering keys according to a set of values. Three kinds of operators are supported: `in`, `notin` and `exists` (only the key identifier). For example:

```
environment in (production, qa)
tier notin (frontend, backend)
partition
!partition
```

- The first example selects all resources with key equal to `environment` and value equal to `production` or `qa`.
- The second example selects all resources with key equal to `tier` and values other than `frontend` and `backend`, and all resources with no labels with the `tier` key.
- The third example selects all resources including a label with key `partition`; no values are checked.
- The fourth example selects all resources without a label with key `partition`; no values are checked.

Similarly the comma separator acts as an AND operator. So filtering resources with a `partition` key (no matter the value) and with `environment` different than `qa` can be achieved using `partition,environment notin (qa)`. The set-based label selector is a general form of equality since `environment=production` is equivalent to `environment in (production)`; similarly for `!=` and `notin`.

Set-based requirements can be mixed with equality-based requirements. For example: `partition in (customerA, customerB),environment!=qa`.

API

LIST and WATCH filtering

LIST and WATCH operations may specify label selectors to filter the sets of objects returned using a query parameter. Both requirements are permitted (presented here as they would appear in a URL query string):

- equality-based requirements: ?
`labelSelector=environment%3Dproduction,tier%3Dfrontend`
- set-based requirements: ?`labelSelector=environment+in+%`
`28production%2Cqa%29%2Ctier+in+%28frontend%29`

Both label selector styles can be used to list or watch resources via a REST client. For example, targeting `apiserver` with `kubectl` and using equality-based one may write:

```
kubectl get pods -l environment=production,tier=frontend
```

or using set-based requirements:

```
kubectl get pods -l 'environment in (production),tier in (frontend)'
```

As already mentioned set-based requirements are more expressive. For instance, they can implement the *OR* operator on values:

```
kubectl get pods -l 'environment in (production, qa)'
```

or restricting negative matching via *exists* operator:

```
kubectl get pods -l 'environment,environment notin (frontend)'
```

Set references in API objects

Some Kubernetes objects, such as [services](#) and [replicationcontrollers](#), also use label selectors to specify sets of other resources, such as [pods](#).

Service and ReplicationController

The set of pods that a service targets is defined with a label selector. Similarly, the population of pods that a replicationcontroller should manage is also defined with a label selector.

Labels selectors for both objects are defined in `json` or `yaml` files using maps, and only equality-based requirement selectors are supported:

```
"selector": {  
    "component": "redis",  
}
```

or

```
selector:  
  component: redis
```

this selector (respectively in `json` or `yaml` format) is equivalent to `component=redis` or `component in (redis)`.

Resources that support set-based requirements

Newer resources, such as [Job](#), [Deployment](#), [ReplicaSet](#), and [DaemonSet](#), support set-based requirements as well.

```
selector:  
  matchLabels:  
    component: redis  
  matchExpressions:  
    - {key: tier, operator: In, values: [cache]}  
    - {key: environment, operator: NotIn, values: [dev]}
```

`matchLabels` is a map of `{key, value}` pairs. A single `{key, value}` in the `matchLabels` map is equivalent to an element of `matchExpressions`, whose key

field is "key", the operator is "In", and the values array contains only "value". matchExpressions is a list of pod selector requirements. Valid operators include In, NotIn, Exists, and DoesNotExist. The values set must be non-empty in the case of In and NotIn. All of the requirements, from both matchLabels and matchExpressions are ANDed together -- they must all be satisfied in order to match.

Selecting sets of nodes

One use case for selecting over labels is to constrain the set of nodes onto which a pod can schedule. See the documentation on [node selection](#) for more information.

Feedback

Was this page helpful?

Yes *No*

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

Last modified December 02, 2020 at 11:59 PM PST: [Update md format to make it more easier to read \(8e0973b22\)](#)

[Edit this page](#) [Create child page](#) [Create an issue](#)

- [Motivation](#)
- [Syntax and character set](#)
- [Label selectors](#)
 - [Equality-based requirement](#)
 - [Set-based requirement](#)
- [API](#)
 - [LIST and WATCH filtering](#)
 - [Set references in API objects](#)

Annotations

You can use Kubernetes annotations to attach arbitrary non-identifying metadata to objects. Clients such as tools and libraries can retrieve this metadata.

Attaching metadata to objects

You can use either labels or annotations to attach metadata to Kubernetes objects. Labels can be used to select objects and to find collections of objects that satisfy certain conditions. In contrast, annotations are not used to identify and select objects. The metadata in an annotation can be small or

large, structured or unstructured, and can include characters not permitted by labels.

Annotations, like labels, are key/value maps:

```
"metadata": {  
  "annotations": {  
    "key1" : "value1",  
    "key2" : "value2"  
  }  
}
```

Here are some examples of information that could be recorded in annotations:

- *Fields managed by a declarative configuration layer. Attaching these fields as annotations distinguishes them from default values set by clients or servers, and from auto-generated fields and fields set by auto-sizing or auto-scaling systems.*
- *Build, release, or image information like timestamps, release IDs, git branch, PR numbers, image hashes, and registry address.*
- *Pointers to logging, monitoring, analytics, or audit repositories.*
- *Client library or tool information that can be used for debugging purposes: for example, name, version, and build information.*
- *User or tool/system provenance information, such as URLs of related objects from other ecosystem components.*
- *Lightweight rollout tool metadata: for example, config or checkpoints.*
- *Phone or pager numbers of persons responsible, or directory entries that specify where that information can be found, such as a team web site.*
- *Directives from the end-user to the implementations to modify behavior or engage non-standard features.*

Instead of using annotations, you could store this type of information in an external database or directory, but that would make it much harder to produce shared client libraries and tools for deployment, management, introspection, and the like.

Syntax and character set

Annotations are key/value pairs. Valid annotation keys have two segments: an optional prefix and name, separated by a slash (/). The name segment is required and must be 63 characters or less, beginning and ending with an alphanumeric character ([a-zA-Z0-9]) with dashes (-), underscores (_), dots (.), and alphanumerics between. The prefix is optional. If specified, the

prefix must be a DNS subdomain: a series of DNS labels separated by dots (.), not longer than 253 characters in total, followed by a slash (/).

If the prefix is omitted, the annotation Key is presumed to be private to the user. Automated system components (e.g. kube-scheduler, kube-controller-manager, kube-apiserver, kubectl, or other third-party automation) which add annotations to end-user objects must specify a prefix.

The kubernetes.io/ and k8s.io/ prefixes are reserved for Kubernetes core components.

For example, here's the configuration file for a Pod that has the annotation imageregistry: https://hub.docker.com/ :

```
apiVersion: v1
kind: Pod
metadata:
  name: annotations-demo
  annotations:
    imageregistry: "https://hub.docker.com/"
spec:
  containers:
  - name: nginx
    image: nginx:1.14.2
    ports:
    - containerPort: 80
```

What's next

Learn more about [Labels and Selectors](#).

Feedback

Was this page helpful?

Yes No

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

Last modified August 05, 2020 at 3:17 AM PST: [Replace special quote characters with normal ones. \(c6a96128c\)](#)

[Edit this page](#) [Create child page](#) [Create an issue](#)

- [Attaching metadata to objects](#)
- [Syntax and character set](#)
- [What's next](#)

Field Selectors

Field selectors let you [select Kubernetes resources](#) based on the value of one or more resource fields. Here are some examples of field selector queries:

- `metadata.name=my-service`
- `metadata.namespace!=default`
- `status.phase=Pending`

This `kubectl` command selects all Pods for which the value of the [`status.phase`](#) field is `Running`:

```
kubectl get pods --field-selector status.phase=Running
```

Note: Field selectors are essentially resource filters. By default, no selectors/filters are applied, meaning that all resources of the specified type are selected. This makes the `kubectl` queries `kubectl get pods` and `kubectl get pods --field-selector ""` equivalent.

Supported fields

Supported field selectors vary by Kubernetes resource type. All resource types support the `metadata.name` and `metadata.namespace` fields. Using unsupported field selectors produces an error. For example:

```
kubectl get ingress --field-selector foo.bar=baz
```

Error from server (BadRequest): Unable to find "ingresses" that match label selector "", field selector "foo.bar=baz": "foo.bar" is not a known field selector: only "metadata.name", "metadata.namespace"

Supported operators

You can use the `=`, `==`, and `!=` operators with field selectors (`=` and `==` mean the same thing). This `kubectl` command, for example, selects all Kubernetes Services that aren't in the default namespace:

```
kubectl get services --all-namespaces --field-selector metadata.namespace!=default
```

Chained selectors

As with [label](#) and other selectors, field selectors can be chained together as a comma-separated list. This `kubectl` command selects all Pods for which the `status.phase` does not equal `Running` and the `spec.restartPolicy` field equals `Always`:

```
kubectl get pods --field-selector=status.phase!=Running,spec.restartPolicy=Always
```

Multiple resource types

You use field selectors across multiple resource types. This `kubectl` command selects all Statefulsets and Services that are not in the default namespace:

```
kubectl get statefulsets,services --all-namespaces --field-selector metadata.namespace!=default
```

Feedback

Was this page helpful?

Yes No

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

Last modified June 08, 2020 at 7:10 PM PST: [workaround for new lines in field selector example \(2f6093bfd\)](#)

[Edit this page](#) [Create child page](#) [Create an issue](#)

- [Supported fields](#)
- [Supported operators](#)
- [Chained selectors](#)
- [Multiple resource types](#)

Recommended Labels

You can visualize and manage Kubernetes objects with more tools than `kubectl` and the dashboard. A common set of labels allows tools to work interoperably, describing objects in a common manner that all tools can understand.

In addition to supporting tooling, the recommended labels describe applications in a way that can be queried.

The metadata is organized around the concept of an application. Kubernetes is not a platform as a service (PaaS) and doesn't have or enforce a formal notion of an application. Instead, applications are informal and described with metadata. The definition of what an application contains is loose.

Note: These are recommended labels. They make it easier to manage applications but aren't required for any core tooling.

Shared labels and annotations share a common prefix: `app.kubernetes.io`. Labels without a prefix are private to users. The shared prefix ensures that shared labels do not interfere with custom user labels.

Labels

In order to take full advantage of using these labels, they should be applied on every resource object.

Key	Description	Example	Type
<code>app.kubernetes.io/name</code>	The name of the application	<code>mysql</code>	string
<code>app.kubernetes.io/instance</code>	A unique name identifying the instance of an application	<code>mysql-abcxyz</code>	string
<code>app.kubernetes.io/version</code>	The current version of the application (e.g., a semantic version, revision hash, etc.)	<code>5.7.21</code>	string
<code>app.kubernetes.io/component</code>	The component within the architecture	<code>database</code>	string
<code>app.kubernetes.io/part-of</code>	The name of a higher level application this one is part of	<code>wordpress</code>	string
<code>app.kubernetes.io/managed-by</code>	The tool being used to manage the operation of an application	<code>helm</code>	string

To illustrate these labels in action, consider the following StatefulSet object:

```
apiVersion: apps/v1
kind: StatefulSet
metadata:
  labels:
    app.kubernetes.io/name: mysql
    app.kubernetes.io/instance: mysql-abcxyz
    app.kubernetes.io/version: "5.7.21"
    app.kubernetes.io/component: database
    app.kubernetes.io/part-of: wordpress
    app.kubernetes.io/managed-by: helm
```

Applications And Instances Of Applications

An application can be installed one or more times into a Kubernetes cluster and, in some cases, the same namespace. For example, `wordpress` can be installed more than once where different websites are different installations of `wordpress`.

The name of an application and the instance name are recorded separately. For example, WordPress has a `app.kubernetes.io/name` of `wordpress` while it has an instance name, represented as `app.kubernetes.io/instance` with a value of `wordpress-abcxyz`. This enables the application and instance of the application to be identifiable. Every instance of an application must have a unique name.

Examples

To illustrate different ways to use these labels the following examples have varying complexity.

A Simple Stateless Service

Consider the case for a simple stateless service deployed using *Deployment* and *Service* objects. The following two snippets represent how the labels could be used in their simplest form.

The *Deployment* is used to oversee the pods running the application itself.

```
apiVersion: apps/v1
kind: Deployment
metadata:
  labels:
    app.kubernetes.io/name: myservice
    app.kubernetes.io/instance: myservice-abcxzy
...
```

The *Service* is used to expose the application.

```
apiVersion: v1
kind: Service
metadata:
  labels:
    app.kubernetes.io/name: myservice
    app.kubernetes.io/instance: myservice-abcxzy
...
```

Web Application With A Database

Consider a slightly more complicated application: a web application (WordPress) using a database (MySQL), installed using Helm. The following snippets illustrate the start of objects used to deploy this application.

The start to the following *Deployment* is used for WordPress:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  labels:
    app.kubernetes.io/name: wordpress
    app.kubernetes.io/instance: wordpress-abcxzy
    app.kubernetes.io/version: "4.9.4"
    app.kubernetes.io/managed-by: helm
    app.kubernetes.io/component: server
    app.kubernetes.io/part-of: wordpress
...
```

The *Service* is used to expose WordPress:

```
apiVersion: v1
kind: Service
metadata:
  labels:
    app.kubernetes.io/name: wordpress
    app.kubernetes.io/instance: wordpress-abcxzy
    app.kubernetes.io/version: "4.9.4"
    app.kubernetes.io/managed-by: helm
    app.kubernetes.io/component: server
    app.kubernetes.io/part-of: wordpress
...
```

MySQL is exposed as a *StatefulSet* with metadata for both it and the larger application it belongs to:

```
apiVersion: apps/v1
kind: StatefulSet
metadata:
  labels:
    app.kubernetes.io/name: mysql
    app.kubernetes.io/instance: mysql-abcxzy
    app.kubernetes.io/version: "5.7.21"
    app.kubernetes.io/managed-by: helm
    app.kubernetes.io/component: database
    app.kubernetes.io/part-of: wordpress
...
```

The *Service* is used to expose MySQL as part of WordPress:

```
apiVersion: v1
kind: Service
metadata:
  labels:
    app.kubernetes.io/name: mysql
    app.kubernetes.io/instance: mysql-abcxzy
    app.kubernetes.io/version: "5.7.21"
    app.kubernetes.io/managed-by: helm
    app.kubernetes.io/component: database
    app.kubernetes.io/part-of: wordpress
...
```

With the MySQL *StatefulSet* and *Service* you'll notice information about both MySQL and Wordpress, the broader application, are included.

Feedback

Was this page helpful?

Yes No

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

Last modified July 10, 2020 at 12:21 PM PST: [Fix the example values in the first examples \(f4d296d1c\)](#)

[Edit this page](#) [Create child page](#) [Create an issue](#)

- [Labels](#)
- [Applications And Instances Of Applications](#)
- [Examples](#)
 - [A Simple Stateless Service](#)
 - [Web Application With A Database](#)

Cluster Architecture

The architectural concepts behind Kubernetes.

[Nodes](#)

[Control Plane-Node Communication](#)

[Controllers](#)

[Cloud Controller Manager](#)

Nodes

Kubernetes runs your workload by placing containers into Pods to run on Nodes. A node may be a virtual or physical machine, depending on the cluster. Each node contains the services necessary to run [Pods](#), managed by the [control plane](#).

Typically you have several nodes in a cluster; in a learning or resource-limited environment, you might have just one.

The [components](#) on a node include the [kubelet](#), a [container runtime](#), and the [kube-proxy](#).

Management

There are two main ways to have Nodes added to the [API server](#):

1. The kubelet on a node self-registers to the control plane
2. You, or another human user, manually add a Node object

After you create a Node object, or the kubelet on a node self-registers, the control plane checks whether the new Node object is valid. For example, if you try to create a Node from the following JSON manifest:

```
{
  "kind": "Node",
  "apiVersion": "v1",
  "metadata": {
    "name": "10.240.79.157",
    "labels": {
      "name": "my-first-k8s-node"
    }
  }
}
```

Kubernetes creates a Node object internally (the representation). Kubernetes checks that a kubelet has registered to the API server that matches the metadata.name field of the Node. If the node is healthy (if all necessary services are running), it is eligible to run a Pod. Otherwise, that node is ignored for any cluster activity until it becomes healthy.

Note:

Kubernetes keeps the object for the invalid Node and continues checking to see whether it becomes healthy.

You, or a [controller](#), must explicitly delete the Node object to stop that health checking.

The name of a Node object must be a valid [DNS subdomain name](#).

Self-registration of Nodes

When the kubelet flag --register-node is true (the default), the kubelet will attempt to register itself with the API server. This is the preferred pattern, used by most distros.

For self-registration, the kubelet is started with the following options:

- `--kubeconfig` - Path to credentials to authenticate itself to the API server.
- `--cloud-provider` - How to talk to a [cloud provider](#) to read metadata about itself.
- `--register-node` - Automatically register with the API server.
- `--register-with-taints` - Register the node with the given list of [taints](#) (comma separated `<key>=<value>:<effect>`).

No-op if register-node is false.

- `--node-ip` - IP address of the node.

- `--node-labels` - [Labels](#) to add when registering the node in the cluster (see label restrictions enforced by the [NodeRestriction admission plugin](#)).
- `--node-status-update-frequency` - Specifies how often kubelet posts node status to master.

When the [Node authorization mode](#) and [NodeRestriction admission plugin](#) are enabled, kubelets are only authorized to create/modify their own Node resource.

Manual Node administration

You can create and modify Node objects using [kubectl](#).

When you want to create Node objects manually, set the kubelet flag `--register-node=false`.

You can modify Node objects regardless of the setting of `--register-node`. For example, you can set labels on an existing Node, or mark it unschedulable.

You can use labels on Nodes in conjunction with node selectors on Pods to control scheduling. For example, you can constrain a Pod to only be eligible to run on a subset of the available nodes.

Marking a node as unschedulable prevents the scheduler from placing new pods onto that Node, but does not affect existing Pods on the Node. This is useful as a preparatory step before a node reboot or other maintenance.

To mark a Node unschedulable, run:

```
kubectl cordon $NODENAME
```

Note: Pods that are part of a [DaemonSet](#) tolerate being run on an unschedulable Node. DaemonSets typically provide node-local services that should run on the Node even if it is being drained of workload applications.

Node status

A Node's status contains the following information:

- [Addresses](#)
- [Conditions](#)
- [Capacity and Allocatable](#)
- [Info](#)

You can use `kubectl` to view a Node's status and other details:

```
kubectl describe node <insert-node-name-here>
```

Each section of the output is described below.

Addressees

The usage of these fields varies depending on your cloud provider or bare metal configuration.

- *HostName*: The hostname as reported by the node's kernel. Can be overridden via the kubelet `--hostname-override` parameter.
- *ExternalIP*: Typically the IP address of the node that is externally routable (available from outside the cluster).
- *InternalIP*: Typically the IP address of the node that is routable only within the cluster.

Conditions

The *conditions* field describes the status of all *Running* nodes. Examples of conditions include:

Node Condition	Description
Ready	True if the node is healthy and ready to accept pods, False if the node is not healthy and is not accepting pods, and Unknown if the node controller has not heard from the node in the last <code>node-monitor-grace-period</code> (default is 40 seconds)
DiskPressure	True if pressure exists on the disk size--that is, if the disk capacity is low; otherwise False
MemoryPressure	True if pressure exists on the node memory--that is, if the node memory is low; otherwise False
PIDPressure	True if pressure exists on the processes--that is, if there are too many processes on the node; otherwise False
NetworkUnavailable	True if the network for the node is not correctly configured, otherwise False

Note: If you use command-line tools to print details of a cordoned Node, the Condition includes `SchedulingDisabled`. `SchedulingDisabled` is not a Condition in the Kubernetes API; instead, cordoned nodes are marked `Unschedulable` in their spec.

The node condition is represented as a JSON object. For example, the following structure describes a healthy node:

```
"conditions": [
  {
    "type": "Ready",
    "status": "True",
    "reason": "KubeletReady",
    "message": "kubelet is posting ready status",
    "lastHeartbeatTime": "2019-06-05T18:38:35Z",
    "lastTransitionTime": "2019-06-05T11:41:27Z"
  }
]
```

If the Status of the Ready condition remains Unknown or False for longer than the pod-eviction-timeout (an argument passed to the [kube-controller-manager](#)), all the Pods on the node are scheduled for deletion by the node controller. The default eviction timeout duration is **five minutes**. In some cases when the node is unreachable, the API server is unable to communicate with the kubelet on the node. The decision to delete the pods cannot be communicated to the kubelet until communication with the API server is re-established. In the meantime, the pods that are scheduled for deletion may continue to run on the partitioned node.

The node controller does not force delete pods until it is confirmed that they have stopped running in the cluster. You can see the pods that might be running on an unreachable node as being in the Terminating or Unknown state. In cases where Kubernetes cannot deduce from the underlying infrastructure if a node has permanently left a cluster, the cluster administrator may need to delete the node object by hand. Deleting the node object from Kubernetes causes all the Pod objects running on the node to be deleted from the API server, and frees up their names.

The node lifecycle controller automatically creates [taints](#) that represent conditions. The scheduler takes the Node's taints into consideration when assigning a Pod to a Node. Pods can also have tolerations which let them tolerate a Node's taints.

See [Taint Nodes by Condition](#) for more details.

Capacity and Allocatable

Describes the resources available on the node: CPU, memory and the maximum number of pods that can be scheduled onto the node.

The fields in the capacity block indicate the total amount of resources that a Node has. The allocatable block indicates the amount of resources on a Node that is available to be consumed by normal Pods.

You may read more about capacity and allocatable resources while learning how to [reserve compute resources](#) on a Node.

Info

Describes general information about the node, such as kernel version, Kubernetes version (kubelet and kube-proxy version), Docker version (if used), and OS name. This information is gathered by Kubelet from the node.

Node controller

The node [controller](#) is a Kubernetes control plane component that manages various aspects of nodes.

The node controller has multiple roles in a node's life. The first is assigning a CIDR block to the node when it is registered (if CIDR assignment is turned on).

The second is keeping the node controller's internal list of nodes up to date with the cloud provider's list of available machines. When running in a cloud environment, whenever a node is unhealthy, the node controller asks the cloud provider if the VM for that node is still available. If not, the node controller deletes the node from its list of nodes.

The third is monitoring the nodes' health. The node controller is responsible for updating the NodeReady condition of NodeStatus to ConditionUnknown when a node becomes unreachable (i.e. the node controller stops receiving heartbeats for some reason, for example due to the node being down), and then later evicting all the pods from the node (using graceful termination) if the node continues to be unreachable. (The default timeouts are 40s to start reporting ConditionUnknown and 5m after that to start evicting pods.) The node controller checks the state of each node every --node-monitor-period seconds.

Heartbeats

Heartbeats, sent by Kubernetes nodes, help determine the availability of a node.

There are two forms of heartbeats: updates of NodeStatus and the [Lease object](#). Each Node has an associated Lease object in the kube-node-lease namespace. Lease is a lightweight resource, which improves the performance of the node heartbeats as the cluster scales.

The kubelet is responsible for creating and updating the NodeStatus and a Lease object.

- *The kubelet updates the NodeStatus either when there is change in status, or if there has been no update for a configured interval. The default interval for NodeStatus updates is 5 minutes (much longer than the 40 second default timeout for unreachable nodes).*
- *The kubelet creates and then updates its Lease object every 10 seconds (the default update interval). Lease updates occur independently from the NodeStatus updates. If the Lease update fails, the kubelet retries with exponential backoff starting at 200 milliseconds and capped at 7 seconds.*

Reliability

In most cases, the node controller limits the eviction rate to --node-eviction-rate (default 0.1) per second, meaning it won't evict pods from more than 1 node per 10 seconds.

The node eviction behavior changes when a node in a given availability zone becomes unhealthy. The node controller checks what percentage of nodes in the zone are unhealthy (NodeReady condition is ConditionUnknown or ConditionFalse) at the same time. If the fraction of unhealthy nodes is at least --unhealthy-zone-threshold (default 0.55) then the eviction rate is reduced: if the cluster is small (i.e. has less than or equal to --large-cluster-size-threshold nodes - default 50) then evictions are stopped,

otherwise the eviction rate is reduced to --secondary-node-eviction-rate (default 0.01) per second. The reason these policies are implemented per availability zone is because one availability zone might become partitioned from the master while the others remain connected. If your cluster does not span multiple cloud provider availability zones, then there is only one availability zone (the whole cluster).

A key reason for spreading your nodes across availability zones is so that the workload can be shifted to healthy zones when one entire zone goes down. Therefore, if all nodes in a zone are unhealthy then the node controller evicts at the normal rate of --node-eviction-rate. The corner case is when all zones are completely unhealthy (i.e. there are no healthy nodes in the cluster). In such a case, the node controller assumes that there's some problem with master connectivity and stops all evictions until some connectivity is restored.

The node controller is also responsible for evicting pods running on nodes with NoExecute taints, unless those pods tolerate that taint. The node controller also adds [taints](#) corresponding to node problems like node unreachable or not ready. This means that the scheduler won't place Pods onto unhealthy nodes.

Caution: `kubectl cordon` marks a node as 'unschedulable', which has the side effect of the service controller removing the node from any LoadBalancer node target lists it was previously eligible for, effectively removing incoming load balancer traffic from the cordoned node(s).

Node capacity

Node objects track information about the Node's resource capacity (for example: the amount of memory available, and the number of CPUs). Nodes that [self register](#) report their capacity during registration. If you [manually](#) add a Node, then you need to set the node's capacity information when you add it.

The Kubernetes [scheduler](#) ensures that there are enough resources for all the Pods on a Node. The scheduler checks that the sum of the requests of containers on the node is no greater than the node's capacity. That sum of requests includes all containers managed by the kubelet, but excludes any containers started directly by the container runtime, and also excludes any processes running outside of the kubelet's control.

Note: If you want to explicitly reserve resources for non-Pod processes, see [reserve resources for system daemons](#).

Node topology

FEATURE STATE: Kubernetes v1.16 [alpha]

If you have enabled the `TopologyManager` [feature gate](#), then the kubelet can use topology hints when making resource assignment decisions. See [Control Topology Management Policies on a Node](#) for more information.

Graceful Node Shutdown

FEATURE STATE: Kubernetes v1.20 [alpha]

If you have enabled the `GracefulNodeShutdown` [feature gate](#), then the kubelet attempts to detect the node system shutdown and terminates pods running on the node. Kubelet ensures that pods follow the normal [pod termination process](#) during the node shutdown.

When the `GracefulNodeShutdown` feature gate is enabled, kubelet uses [systemd inhibitor locks](#) to delay the node shutdown with a given duration. During a shutdown kubelet terminates pods in two phases:

1. Terminate regular pods running on the node.
2. Terminate [critical pods](#) running on the node.

Graceful Node Shutdown feature is configured with two [Kubelet Configuration options](#):

- `ShutdownGracePeriod`:
 - Specifies the total duration that the node should delay the shutdown by. This is the total grace period for pod termination for both regular and [critical pods](#).
- `ShutdownGracePeriodCriticalPods`:
 - Specifies the duration used to terminate [critical pods](#) during a node shutdown. This should be less than `ShutdownGracePeriod`.

For example, if `ShutdownGracePeriod=30s`, and `ShutdownGracePeriodCriticalPods=10s`, kubelet will delay the node shutdown by 30 seconds. During the shutdown, the first 20 (30-10) seconds would be reserved for gracefully terminating normal pods, and the last 10 seconds would be reserved for terminating [critical pods](#).

What's next

- Learn about the [components](#) that make up a node.
- Read the [API definition for Node](#).
- Read the [Node](#) section of the architecture design document.
- Read about [taints and tolerations](#).

Feedback

Was this page helpful?

Yes No

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

Last modified October 07, 2020 at 7:16 PM PST: [Revise cluster management task \(59dcd57cc\)](#)

[Edit this page](#) [Create child page](#) [Create an issue](#)

- [Management](#)
 - [Self-registration of Nodes](#)
 - [Manual Node administration](#)
- [Node status](#)
 - [Addresses](#)
 - [Conditions](#)
 - [Capacity and Allocatable](#)
 - [Info](#)
 - [Node controller](#)
 - [Node capacity](#)
- [Node topology](#)
- [Graceful Node Shutdown](#)
- [What's next](#)

Control Plane-Node Communication

This document catalogs the communication paths between the control plane (really the apiserver) and the Kubernetes cluster. The intent is to allow users to customize their installation to harden the network configuration such that the cluster can be run on an untrusted network (or on fully public IPs on a cloud provider).

Node to Control Plane

Kubernetes has a "hub-and-spoke" API pattern. All API usage from nodes (or the pods they run) terminate at the apiserver (none of the other control plane components are designed to expose remote services). The apiserver is configured to listen for remote connections on a secure HTTPS port (typically 443) with one or more forms of client [authentication](#) enabled. One or more forms of [authorization](#) should be enabled, especially if [anonymous requests](#) or [service account tokens](#) are allowed.

Nodes should be provisioned with the public root certificate for the cluster such that they can connect securely to the apiserver along with valid client credentials. A good approach is that the client credentials provided to the kubelet are in the form of a client certificate. See [kubelet TLS bootstrapping](#) for automated provisioning of kubelet client certificates.

Pods that wish to connect to the apiserver can do so securely by leveraging a service account so that Kubernetes will automatically inject the public root

certificate and a valid bearer token into the pod when it is instantiated. The kubernetes service (in all namespaces) is configured with a virtual IP address that is redirected (via kube-proxy) to the HTTPS endpoint on the apiserver.

The control plane components also communicate with the cluster apiserver over the secure port.

As a result, the default operating mode for connections from the nodes and pods running on the nodes to the control plane is secured by default and can run over untrusted and/or public networks.

Control Plane to node

There are two primary communication paths from the control plane (apiserver) to the nodes. The first is from the apiserver to the kubelet process which runs on each node in the cluster. The second is from the apiserver to any node, pod, or service through the apiserver's proxy functionality.

apiserver to kubelet

The connections from the apiserver to the kubelet are used for:

- Fetching logs for pods.
- Attaching (through kubectl) to running pods.
- Providing the kubelet's port-forwarding functionality.

*These connections terminate at the kubelet's HTTPS endpoint. By default, the apiserver does not verify the kubelet's serving certificate, which makes the connection subject to man-in-the-middle attacks, and **unsafe** to run over untrusted and/or public networks.*

To verify this connection, use the `--kubelet-certificate-authority` flag to provide the apiserver with a root certificate bundle to use to verify the kubelet's serving certificate.

If that is not possible, use [SSH tunneling](#) between the apiserver and kubelet if required to avoid connecting over an untrusted or public network.

Finally, [Kubelet authentication and/or authorization](#) should be enabled to secure the kubelet API.

apiserver to nodes, pods, and services

The connections from the apiserver to a node, pod, or service default to plain HTTP connections and are therefore neither authenticated nor encrypted. They can be run over a secure HTTPS connection by prefixing https: to the node, pod, or service name in the API URL, but they will not validate the certificate provided by the HTTPS endpoint nor provide client credentials so while the connection will be encrypted, it will not provide any

guarantees of integrity. These connections **are not currently safe** to run over untrusted and/or public networks.

SSH tunnels

Kubernetes supports SSH tunnels to protect the control plane to nodes communication paths. In this configuration, the apiserver initiates an SSH tunnel to each node in the cluster (connecting to the ssh server listening on port 22) and passes all traffic destined for a kubelet, node, pod, or service through the tunnel. This tunnel ensures that the traffic is not exposed outside of the network in which the nodes are running.

SSH tunnels are currently deprecated so you shouldn't opt to use them unless you know what you are doing. The Konnectivity service is a replacement for this communication channel.

Konnectivity service

FEATURE STATE: Kubernetes v1.18 [beta]

As a replacement to the SSH tunnels, the Konnectivity service provides TCP level proxy for the control plane to cluster communication. The Konnectivity service consists of two parts: the Konnectivity server and the Konnectivity agents, running in the control plane network and the nodes network respectively. The Konnectivity agents initiate connections to the Konnectivity server and maintain the network connections. After enabling the Konnectivity service, all control plane to nodes traffic goes through these connections.

Follow the [Konnectivity service task](#) to set up the Konnectivity service in your cluster.

Feedback

Was this page helpful?

Yes No

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

Last modified October 22, 2020 at 2:24 PM PST: [Fix links in concepts section \(070023b24\)](#)

[Edit this page](#) [Create child page](#) [Create an issue](#)

- [Node to Control Plane](#)
- [Control Plane to node](#)
 - [apiserver to kubelet](#)
 - [apiserver to nodes, pods, and services](#)
 - [SSH tunnels](#)
 - [Konnectivity service](#)

Controllers

In robotics and automation, a control loop is a non-terminating loop that regulates the state of a system.

Here is one example of a control loop: a thermostat in a room.

When you set the temperature, that's telling the thermostat about your desired state. The actual room temperature is the current state. The thermostat acts to bring the current state closer to the desired state, by turning equipment on or off.

In Kubernetes, controllers are control loops that watch the state of your [cluster](#), then make or request changes where needed. Each controller tries to move the current cluster state closer to the desired state.

Controller pattern

A controller tracks at least one Kubernetes resource type. These [objects](#) have a spec field that represents the desired state. The controller(s) for that resource are responsible for making the current state come closer to that desired state.

The controller might carry the action out itself; more commonly, in Kubernetes, a controller will send messages to the [API server](#) that have useful side effects. You'll see examples of this below.

Control via API server

The [Job](#) controller is an example of a Kubernetes built-in controller. Built-in controllers manage state by interacting with the cluster API server.

Job is a Kubernetes resource that runs a [Pod](#), or perhaps several Pods, to carry out a task and then stop.

(Once [scheduled](#), Pod objects become part of the desired state for a kubelet).

When the Job controller sees a new task it makes sure that, somewhere in your cluster, the kubelets on a set of Nodes are running the right number of Pods to get the work done. The Job controller does not run any Pods or containers itself. Instead, the Job controller tells the API server to create or remove Pods. Other components in the [control plane](#) act on the new information (there are new Pods to schedule and run), and eventually the work is done.

After you create a new Job, the desired state is for that Job to be completed. The Job controller makes the current state for that Job be nearer to your desired state: creating Pods that do the work you wanted for that Job, so that the Job is closer to completion.

Controllers also update the objects that configure them. For example: once the work is done for a Job, the Job controller updates that Job object to mark it Finished.

(This is a bit like how some thermostats turn a light off to indicate that your room is now at the temperature you set).

Direct control

By contrast with Job, some controllers need to make changes to things outside of your cluster.

For example, if you use a control loop to make sure there are enough [Nodes](#) in your cluster, then that controller needs something outside the current cluster to set up new Nodes when needed.

Controllers that interact with external state find their desired state from the API server, then communicate directly with an external system to bring the current state closer in line.

(There actually is a [controller](#) that horizontally scales the nodes in your cluster.)

The important point here is that the controller makes some change to bring about your desired state, and then reports current state back to your cluster's API server. Other control loops can observe that reported data and take their own actions.

In the thermostat example, if the room is very cold then a different controller might also turn on a frost protection heater. With Kubernetes clusters, the control plane indirectly works with IP address management tools, storage services, cloud provider APIs, and other services by [extending Kubernetes](#) to implement that.

Desired versus current state

Kubernetes takes a cloud-native view of systems, and is able to handle constant change.

Your cluster could be changing at any point as work happens and control loops automatically fix failures. This means that, potentially, your cluster never reaches a stable state.

As long as the controllers for your cluster are running and able to make useful changes, it doesn't matter if the overall state is or is not stable.

Design

As a tenet of its design, Kubernetes uses lots of controllers that each manage a particular aspect of cluster state. Most commonly, a particular control loop (controller) uses one kind of resource as its desired state, and has a different kind of resource that it manages to make that desired state happen. For example, a controller for Jobs tracks Job objects (to discover new work) and Pod objects (to run the Jobs, and then to see when the work is finished). In this case something else creates the Jobs, whereas the Job controller creates Pods.

It's useful to have simple controllers rather than one, monolithic set of control loops that are interlinked. Controllers can fail, so Kubernetes is designed to allow for that.

Note:

There can be several controllers that create or update the same kind of object. Behind the scenes, Kubernetes controllers make sure that they only pay attention to the resources linked to their controlling resource.

For example, you can have Deployments and Jobs; these both create Pods. The Job controller does not delete the Pods that your Deployment created, because there is information ([labels](#)) the controllers can use to tell those Pods apart.

Ways of running controllers

Kubernetes comes with a set of built-in controllers that run inside the [kube-controller-manager](#). These built-in controllers provide important core behaviors.

The Deployment controller and Job controller are examples of controllers that come as part of Kubernetes itself ("built-in" controllers). Kubernetes lets you run a resilient control plane, so that if any of the built-in controllers were to fail, another part of the control plane will take over the work.

You can find controllers that run outside the control plane, to extend Kubernetes. Or, if you want, you can write a new controller yourself. You can run your own controller as a set of Pods, or externally to Kubernetes. What fits best will depend on what that particular controller does.

What's next

- Read about the [Kubernetes control plane](#)
- Discover some of the basic [Kubernetes objects](#)
- Learn more about the [Kubernetes API](#)
- If you want to write your own controller, see [Extension Patterns](#) in Extending Kubernetes.

Feedback

Was this page helpful?

Yes No

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

Last modified October 07, 2020 at 7:16 PM PST: [Revise cluster management task \(59dcd57cc\)](#)

[Edit this page](#) [Create child page](#) [Create an issue](#)

- [Controller pattern](#)
 - [Control via API server](#)
 - [Direct control](#)
- [Desired versus current state](#)
- [Design](#)
- [Ways of running controllers](#)
- [What's next](#)

Cloud Controller Manager

FEATURE STATE: Kubernetes v1.11 [beta]

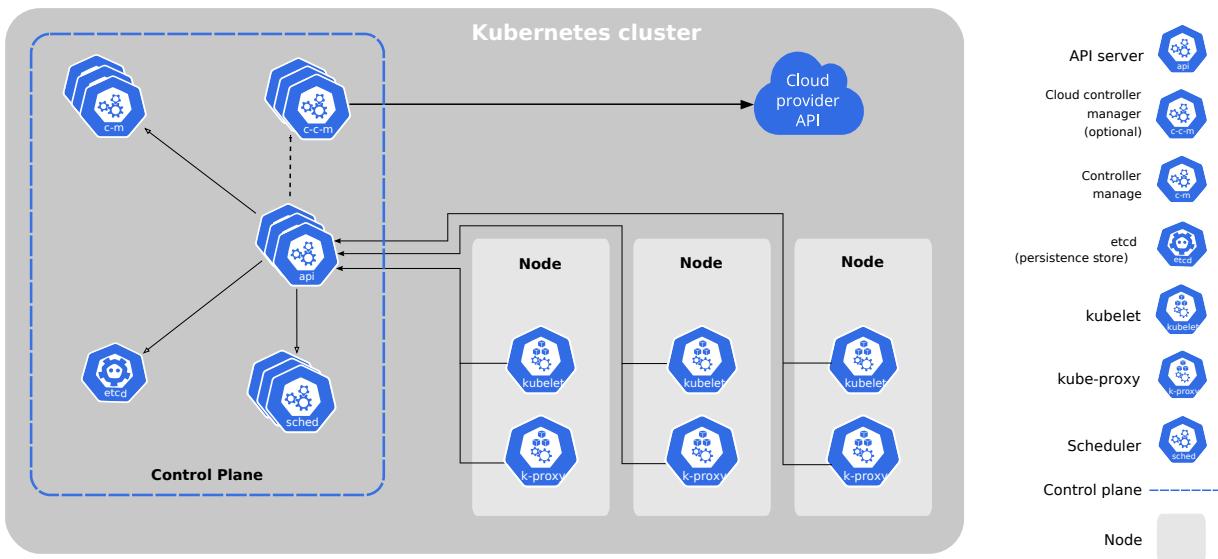
Cloud infrastructure technologies let you run Kubernetes on public, private, and hybrid clouds. Kubernetes believes in automated, API-driven infrastructure without tight coupling between components.

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 just 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.

The cloud-controller-manager is structured using a plugin mechanism that allows different cloud providers to integrate their platforms with Kubernetes.

Design



The cloud controller manager runs in the control plane as a replicated set of processes (usually, these are containers in Pods). Each cloud-controller-manager implements multiple [controllers](#) in a single process.

Note: You can also run the cloud controller manager as a Kubernetes [addon](#) rather than as part of the control plane.

Cloud controller manager functions

The controllers inside the cloud controller manager include:

Node controller

The node controller is responsible for creating [Node](#) objects when new servers are created in your cloud infrastructure. The node controller obtains information about the hosts running inside your tenancy with the cloud provider. The node controller performs the following functions:

1. Initialize a Node object for each server that the controller discovers through the cloud provider API.
2. Annotating and labelling the Node object with cloud-specific information, such as the region the node is deployed into and the resources (CPU, memory, etc) that it has available.
3. Obtain the node's hostname and network addresses.
4. Verifying the node's health. In case a node becomes unresponsive, this controller checks with your cloud provider's API to see if the server has been deactivated / deleted / terminated. If the node has been deleted from the cloud, the controller deletes the Node object from your Kubernetes cluster.

Some cloud provider implementations split this into a node controller and a separate node lifecycle controller.

Route controller

The route controller is responsible for configuring routes in the cloud appropriately so that containers on different nodes in your Kubernetes cluster can communicate with each other.

Depending on the cloud provider, the route controller might also allocate blocks of IP addresses for the Pod network.

Service controller

[Services](#) integrate with cloud infrastructure components such as managed load balancers, IP addresses, network packet filtering, and target health checking. The service controller interacts with your cloud provider's APIs to set up load balancers and other infrastructure components when you declare a Service resource that requires them.

Authorization

This section breaks down the access that the cloud controller managers requires on various API objects, in order to perform its operations.

Node controller

The Node controller only works with Node objects. It requires full access to read and modify Node objects.

v1/Node:

- *Get*
- *List*
- *Create*
- *Update*
- *Patch*
- *Watch*
- *Delete*

Route controller

The route controller listens to Node object creation and configures routes appropriately. It requires Get access to Node objects.

v1/Node:

- *Get*

Service controller

The service controller listens to Service object Create, Update and Delete events and then configures Endpoints for those Services appropriately.

To access Services, it requires List, and Watch access. To update Services, it requires Patch and Update access.

To set up Endpoints resources for the Services, it requires access to Create, List, Get, Watch, and Update.

v1/Service:

- *List*
- *Get*
- *Watch*
- *Patch*
- *Update*

Others

The implementation of the core of the cloud controller manager requires access to create Event objects, and to ensure secure operation, it requires access to create ServiceAccounts.

v1/Event:

- *Create*
- *Patch*
- *Update*

v1/ServiceAccount:

- *Create*

The [RBAC ClusterRole](#) for the cloud controller manager looks like:

```
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRole
metadata:
  name: cloud-controller-manager
rules:
- apiGroups:
  - ""
    resources:
    - events
    verbs:
    - create
    - patch
    - update
- apiGroups:
  - ""
    resources:
    - nodes
    verbs:
    - "*"
- apiGroups:
  - ""
    resources:
    - nodes/status
    verbs:
    - patch
- apiGroups:
  - ""
    resources:
    - services
    verbs:
    - list
    - patch
    - update
    - watch
- apiGroups:
  - ""
    resources:
    - serviceaccounts
    verbs:
    - create
- apiGroups:
  - ""
    resources:
    - persistentvolumes
    verbs:
    - get
    - list
    - update
    - watch
- apiGroups:
```

```
-- ""  
resources:  
- endpoints  
verbs:  
- create  
- get  
- list  
- watch  
- update
```

What's next

[Cloud Controller Manager Administration](#) has instructions on running and managing the cloud controller manager.

Want to know how to implement your own cloud controller manager, or extend an existing project?

The cloud controller manager uses Go interfaces to allow implementations from any cloud to be plugged in. Specifically, it uses the `CloudProvider` interface defined in [cloud.go](#) from [kubernetes/cloud-provider](#).

The implementation of the shared controllers highlighted in this document (`Node`, `Route`, and `Service`), and some scaffolding along with the shared `cloudprovider` interface, is part of the Kubernetes core. Implementations specific to cloud providers are outside the core of Kubernetes and implement the `CloudProvider` interface.

For more information about developing plugins, see [Developing Cloud Controller Manager](#).

Feedback

Was this page helpful?

Yes No

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

Last modified August 28, 2020 at 4:49 PM PST: [Update Kubernetes architectural diagram \(#23138\) \(94b2f857b\)](#)

[Edit this page](#) [Create child page](#) [Create an issue](#)

- [Design](#)
- [Cloud controller manager functions](#)
 - [Node controller](#)
 - [Route controller](#)
 - [Service controller](#)
- [Authorization](#)
 - [Node controller](#)

- [Route controller](#)
- [Service controller](#)
- [Others](#)
- [What's next](#)

Containers

Technology for packaging an application along with its runtime dependencies.

Each container that you run is repeatable; the standardization from having dependencies included means that you get the same behavior wherever you run it.

Containers decouple applications from underlying host infrastructure. This makes deployment easier in different cloud or OS environments.

Container images

A [container image](#) is a ready-to-run software package, containing everything needed to run an application: the code and any runtime it requires, application and system libraries, and default values for any essential settings.

By design, a container is immutable: you cannot change the code of a container that is already running. If you have a containerized application and want to make changes, you need to build a new image that includes the change, then recreate the container to start from the updated image.

Container runtimes

The container runtime is the software that is responsible for running containers.

Kubernetes supports several container runtimes: [Docker](#), [containerd](#), [CRI-O](#), and any implementation of the [Kubernetes CRI \(Container Runtime Interface\)](#).

What's next

- Read about [container images](#)
- Read about [Pods](#)

Images

A container image represents binary data that encapsulates an application and all its software dependencies. Container images are executable software

bundles that can run standalone and that make very well defined assumptions about their runtime environment.

You typically create a container image of your application and push it to a registry before referring to it in a [Pod](#)

This page provides an outline of the container image concept.

Image names

Container images are usually given a name such as `pause`, `example`/`mycontainer`, or `kube-apiserver`. Images can also include a registry hostname; for example: `fictional.registry.example/imagename`, and possibly a port number as well; for example: `fictional.registry.example:10443/imagename`.

If you don't specify a registry hostname, Kubernetes assumes that you mean the Docker public registry.

After the image name part you can add a tag (as also using with commands such as `docker` and `podman`). Tags let you identify different versions of the same series of images.

Image tags consist of lowercase and uppercase letters, digits, underscores (`_`), periods (`.`), and dashes (`-`).

There are additional rules about where you can place the separator characters (`_`, `-`, and `.`) inside an image tag.

If you don't specify a tag, Kubernetes assumes you mean the tag `latest`.

Caution:

You should avoid using the `latest` tag when deploying containers in production, as it is harder to track which version of the image is running and more difficult to roll back to a working version.

Instead, specify a meaningful tag such as `v1.42.0`.

Updating images

The default pull policy is `IfNotPresent` which causes the [kubelet](#) to skip pulling an image if it already exists. If you would like to always force a pull, you can do one of the following:

- set the `imagePullPolicy` of the container to `Always`.
- omit the `imagePullPolicy` and use `:latest` as the tag for the image to use.
- omit the `imagePullPolicy` and the tag for the image to use.
- enable the [`AlwaysPullImages`](#) admission controller.

When `imagePullPolicy` is defined without a specific value, it is also set to `Always`.

Multi-architecture images with image indexes

As well as providing binary images, a container registry can also serve a [container image index](#). An image index can point to multiple [image manifests](#) for architecture-specific versions of a container. The idea is that you can have a name for an image (for example: `pause`, `example/mycontainer`, `kube-apiserver`) and allow different systems to fetch the right binary image for the machine architecture they are using.

Kubernetes itself typically names container images with a suffix `-$(ARCH)`. For backward compatibility, please generate the older images with suffixes. The idea is to generate say `pause` image which has the manifest for all the arch(es) and say `pause-amd64` which is backwards compatible for older configurations or YAML files which may have hard coded the images with suffixes.

Using a private registry

Private registries may require keys to read images from them. Credentials can be provided in several ways:

- Configuring Nodes to Authenticate to a Private Registry
 - all pods can read any configured private registries
 - requires node configuration by cluster administrator
- Pre-pulled Images
 - all pods can use any images cached on a node
 - requires root access to all nodes to setup
- Specifying ImagePullSecrets on a Pod
 - only pods which provide own keys can access the private registry
- Vendor-specific or local extensions
 - if you're using a custom node configuration, you (or your cloud provider) can implement your mechanism for authenticating the node to the container registry.

These options are explained in more detail below.

Configuring nodes to authenticate to a private registry

If you run Docker on your nodes, you can configure the Docker container runtime to authenticate to a private container registry.

This approach is suitable if you can control node configuration.

Note: Default Kubernetes only supports the `auths` and `HttpHeaders` section in Docker configuration. Docker credential helpers (`credHelpers` or `credsStore`) are not supported.

Docker stores keys for private registries in the \$HOME/.dockercfg or \$HOME/.docker/config.json file. If you put the same file in the search paths list below, kubelet uses it as the credential provider when pulling images.

- {--root-dir:-/var/lib/kubelet}/config.json
- {cwd of kubelet}/config.json
- \${HOME}/.docker/config.json
- ./docker/config.json
- {--root-dir:-/var/lib/kubelet}/.dockercfg
- {cwd of kubelet}/.dockercfg
- \${HOME}/.dockercfg
- ./dockercfg

Note: You may have to set HOME=/root explicitly in the environment of the kubelet process.

Here are the recommended steps to configuring your nodes to use a private registry. In this example, run these on your desktop/laptop:

1. Run `docker login [server]` for each set of credentials you want to use. This updates \$HOME/.docker/config.json on your PC.
2. View \$HOME/.docker/config.json in an editor to ensure it contains just the credentials you want to use.
3. Get a list of your nodes; for example:
 - if you want the names: `nodes=$(kubectl get nodes -o jsonpath='{range.items[*].metadata}{.name} {end}')`
 - if you want to get the IP addresses: `nodes=$(kubectl get nodes -o jsonpath='{range.items[*].status.addresses[?(@.type=="ExternalIP")]}{.address} {end}')`
4. Copy your local .docker/config.json to one of the search paths list above.
 - for example, to test this out: `for n in $nodes; do scp ~/docker/config.json root@"$n":/var/lib/kubelet/config.json; done`

Note: For production clusters, use a configuration management tool so that you can apply this setting to all the nodes where you need it.

Verify by creating a Pod that uses a private image; for example:

```
kubectl apply -f - <<EOF
apiVersion: v1
kind: Pod
metadata:
  name: private-image-test-1
spec:
  containers:
    - name: uses-private-image
      image: $PRIVATE_IMAGE_NAME
      imagePullPolicy: Always
      command: [ "echo", "SUCCESS" ]
EOF
```

```
pod/private-image-test-1 created
```

If everything is working, then, after a few moments, you can run:

```
kubectl logs private-image-test-1
```

and see that the command outputs:

```
SUCCESS
```

If you suspect that the command failed, you can run:

```
kubectl describe pods/private-image-test-1 | grep 'Failed'
```

In case of failure, the output is similar to:

```
Fri, 26 Jun 2015 15:36:13 -0700      Fri, 26 Jun 2015 15:39:13  
-0700      19      {kubelet node-i2hq}      spec.containers{uses-  
private-image}      failed      Failed to pull image "user/  
privaterepo:v1": Error: image user/privaterepo:v1 not found
```

You must ensure all nodes in the cluster have the same `.docker/config.json`. Otherwise, pods will run on some nodes and fail to run on others. For example, if you use node autoscaling, then each instance template needs to include the `.docker/config.json` or mount a drive that contains it.

All pods will have read access to images in any private registry once private registry keys are added to the `.docker/config.json`.

Pre-pulled images

Note: This approach is suitable if you can control node configuration. It will not work reliably if your cloud provider manages nodes and replaces them automatically.

By default, the kubelet tries to pull each image from the specified registry. However, if the `imagePullPolicy` property of the container is set to `IfNotPresent` or `Never`, then a local image is used (preferentially or exclusively, respectively).

If you want to rely on pre-pulled images as a substitute for registry authentication, you must ensure all nodes in the cluster have the same pre-pulled images.

This can be used to preload certain images for speed or as an alternative to authenticating to a private registry.

All pods will have read access to any pre-pulled images.

Specifying `imagePullSecrets` on a Pod

Note: This is the recommended approach to run containers based on images in private registries.

Kubernetes supports specifying container image registry keys on a Pod.

Creating a Secret with a Docker config

Run the following command, substituting the appropriate uppercase values:

```
kubectl create secret docker-registry <name> --docker-server=DOCKER_REGISTRY_SERVER --docker-username=DOCKER_USER --docker-password=DOCKER_PASSWORD --docker-email=DOCKER_EMAIL
```

If you already have a Docker credentials file then, rather than using the above command, you can import the credentials file as a Kubernetes [Secrets](#).

[Create a Secret based on existing Docker credentials](#) explains how to set this up.

This is particularly useful if you are using multiple private container registries, as `kubectl create secret docker-registry` creates a Secret that only works with a single private registry.

Note: Pods can only reference image pull secrets in their own namespace, so this process needs to be done one time per namespace.

Referring to an `imagePullSecrets` on a Pod

Now, you can create pods which reference that secret by adding an `imagePullSecrets` section to a Pod definition.

For example:

```
cat <<EOF > pod.yaml
apiVersion: v1
kind: Pod
metadata:
  name: foo
  namespace: awesomeapps
spec:
  containers:
    - name: foo
      image: janedoe/awesomeapp:v1
  imagePullSecrets:
    - name: myregistrykey
EOF
```

```
cat <<EOF >> ./kustomization.yaml
resources:
```

```
- pod.yaml  
EOF
```

This needs to be done for each pod that is using a private registry.

However, setting of this field can be automated by setting the `imagePullSecrets` in a [ServiceAccount](#) resource.

Check [Add ImagePullSecrets to a Service Account](#) for detailed instructions.

You can use this in conjunction with a per-node `.docker/config.json`. The credentials will be merged.

Use cases

There are a number of solutions for configuring private registries. Here are some common use cases and suggested solutions.

1. Cluster running only non-proprietary (e.g. open-source) images. No need to hide images.
 - Use public images on the Docker hub.
 - No configuration required.
 - Some cloud providers automatically cache or mirror public images, which improves availability and reduces the time to pull images.
2. Cluster running some proprietary images which should be hidden to those outside the company, but visible to all cluster users.
 - Use a hosted private [Docker registry](#).
 - It may be hosted on the [Docker Hub](#), or elsewhere.
 - Manually configure `.docker/config.json` on each node as described above.
 - Or, run an internal private registry behind your firewall with open read access.
 - No Kubernetes configuration is required.
 - Use a hosted container image registry service that controls image access
 - It will work better with cluster autoscaling than manual node configuration.
 - Or, on a cluster where changing the node configuration is inconvenient, use `imagePullSecrets`.
3. Cluster with proprietary images, a few of which require stricter access control.
 - Ensure [AlwaysPullImages admission controller](#) is active. Otherwise, all Pods potentially have access to all images.
 - Move sensitive data into a "Secret" resource, instead of packaging it in an image.
4. A multi-tenant cluster where each tenant needs own private registry.
 - Ensure [AlwaysPullImages admission controller](#) is active. Otherwise, all Pods of all tenants potentially have access to all images.
 - Run a private registry with authorization required.

- Generate registry credential for each tenant, put into secret, and populate secret to each tenant namespace.
- The tenant adds that secret to `imagePullSecrets` of each namespace.

If you need access to multiple registries, you can create one secret for each registry. Kubelet will merge any `imagePullSecrets` into a single virtual `.docker/config.json`

What's next

- Read the [OCI Image Manifest Specification](#)

Feedback

Was this page helpful?

Yes No

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

Last modified November 30, 2020 at 1:38 PM PST: [Wrong participate \(0c858f881\)](#)

[Edit this page](#) [Create child page](#) [Create an issue](#)

- [Image names](#)
- [Updating images](#)
- [Multi-architecture images with image indexes](#)
- [Using a private registry](#)
 - [Configuring nodes to authenticate to a private registry](#)
 - [Pre-pulled images](#)
 - [Specifying imagePullSecrets on a Pod](#)
- [Use cases](#)
- [What's next](#)

Container Environment

This page describes the resources available to Containers in the Container environment.

Container environment

The Kubernetes Container environment provides several important resources to Containers:

- A filesystem, which is a combination of an [image](#) and one or more [volumes](#).

- Information about the Container itself.
- Information about other objects in the cluster.

Container information

The hostname of a Container is the name of the Pod in which the Container is running. It is available through the `hostname` command or the [gethostname](#) function call in `libc`.

The Pod name and namespace are available as environment variables through the [downward API](#).

User defined environment variables from the Pod definition are also available to the Container, as are any environment variables specified statically in the Docker image.

Cluster information

A list of all services that were running when a Container was created is available to that Container as environment variables. Those environment variables match the syntax of Docker links.

For a service named `foo` that maps to a Container named `bar`, the following variables are defined:

`FOO_SERVICE_HOST`=<the host the service is running on>
`FOO_SERVICE_PORT`=<the port the service is running on>

Services have dedicated IP addresses and are available to the Container via DNS, if [DNS addon](#) is enabled.Â

What's next

- Learn more about [Container lifecycle hooks](#).
- Get hands-on experience [attaching handlers to Container lifecycle events](#).

Feedback

Was this page helpful?

Yes No

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

Last modified July 17, 2020 at 4:11 PM PST: [Fix links in concepts section \(2\)](#) ([c8f470487](#))

- [Container environment](#)
 - [Container information](#)
 - [Cluster information](#)
- [What's next](#)

Runtime Class

FEATURE STATE: Kubernetes v1.20 [stable]

This page describes the `RuntimeClass` resource and runtime selection mechanism.

`RuntimeClass` is a feature for selecting the container runtime configuration. The container runtime configuration is used to run a Pod's containers.

Motivation

You can set a different `RuntimeClass` between different Pods to provide a balance of performance versus security. For example, if part of your workload deserves a high level of information security assurance, you might choose to schedule those Pods so that they run in a container runtime that uses hardware virtualization. You'd then benefit from the extra isolation of the alternative runtime, at the expense of some additional overhead.

You can also use `RuntimeClass` to run different Pods with the same container runtime but with different settings.

Setup

Ensure the `RuntimeClass` feature gate is enabled (it is by default). See [Feature Gates](#) for an explanation of enabling feature gates. The `RuntimeClass` feature gate must be enabled on apiservers and kubelets.

1. Configure the CRI implementation on nodes (runtime dependent)
2. Create the corresponding `RuntimeClass` resources

1. Configure the CRI implementation on nodes

The configurations available through `RuntimeClass` are Container Runtime Interface (CRI) implementation dependent. See the corresponding documentation ([below](#)) for your CRI implementation for how to configure.

Note: `RuntimeClass` assumes a homogeneous node configuration across the cluster by default (which means that all nodes are configured the same way with respect to container runtimes). To support heterogenous node configurations, see [Scheduling](#) below.

The configurations have a corresponding `handler` name, referenced by the `RuntimeClass`. The handler must be a valid DNS 1123 label (alpha-numeric + - characters).

2. Create the corresponding `RuntimeClass` resources

The configurations setup in step 1 should each have an associated `handler` name, which identifies the configuration. For each handler, create a corresponding `RuntimeClass` object.

The `RuntimeClass` resource currently only has 2 significant fields: the `RuntimeClass` name (`metadata.name`) and the handler (`handler`). The object definition looks like this:

```
apiVersion: node.k8s.io/v1 # RuntimeClass is defined in the
node.k8s.io API group
kind: RuntimeClass
metadata:
  name: myclass # The name the RuntimeClass will be referenced
by
  # RuntimeClass is a non-namespaced resource
handler: myconfiguration # The name of the corresponding CRI
configuration
```

The name of a `RuntimeClass` object must be a valid [DNS subdomain name](#).

Note: It is recommended that `RuntimeClass` write operations (create/update/patch/delete) be restricted to the cluster administrator. This is typically the default. See [Authorization Overview](#) for more details.

Usage

Once `RuntimeClasses` are configured for the cluster, using them is very simple. Specify a `runtimeClassName` in the Pod spec. For example:

```
apiVersion: v1
kind: Pod
metadata:
  name: mypod
spec:
  runtimeClassName: myclass
  # ...
```

This will instruct the Kubelet to use the named `RuntimeClass` to run this pod. If the named `RuntimeClass` does not exist, or the CRI cannot run the corresponding handler, the pod will enter the Failed terminal [phase](#). Look for a corresponding [event](#) for an error message.

If no `runtimeClassName` is specified, the default `RuntimeHandler` will be used, which is equivalent to the behavior when the `RuntimeClass` feature is disabled.

CRI Configuration

For more details on setting up CRI runtimes, see [CRI installation](#).

dockershim

Kubernetes built-in dockershim CRI does not support runtime handlers.

containerd

Runtime handlers are configured through containerd's configuration at `/etc/containerd/config.toml`. Valid handlers are configured under the `runtimes` section:

```
[plugins.cri.containerd.runtimes.${HANDLER_NAME}]
```

See containerd's config documentation for more details: <https://github.com/containerd/cri/blob/master/docs/config.md>

CRI-O

Runtime handlers are configured through CRI-O's configuration at `/etc/crio/crio.conf`. Valid handlers are configured under the [crio.runtime table](#):

```
[crio.runtime.runtimes.${HANDLER_NAME}]
runtime_path = "${PATH_TO_BINARY}"
```

See CRI-O's [config documentation](#) for more details.

Scheduling

FEATURE STATE: Kubernetes v1.16 [beta]

As of Kubernetes v1.16, `RuntimeClass` includes support for heterogenous clusters through its `scheduling` fields. Through the use of these fields, you can ensure that pods running with this `RuntimeClass` are scheduled to nodes that support it. To use the scheduling support, you must have the [RuntimeClass admission controller](#) enabled (the default, as of 1.16).

To ensure pods land on nodes supporting a specific `RuntimeClass`, that set of nodes should have a common label which is then selected by the `runtimeclasses.scheduling.nodeSelector` field. The `RuntimeClass`'s `nodeSelector` is merged with the pod's `nodeSelector` in admission, effectively taking the intersection of the set of nodes selected by each. If there is a conflict, the pod will be rejected.

If the supported nodes are tainted to prevent other `RuntimeClass` pods from running on the node, you can add tolerations to the `RuntimeClass`. As with the `nodeSelector`, the tolerations are merged with the pod's tolerations in admission, effectively taking the union of the set of nodes tolerated by each.

To learn more about configuring the node selector and tolerations, see [Assigning Pods to Nodes](#).

Pod Overhead

FEATURE STATE: Kubernetes v1.18 [beta]

You can specify overhead resources that are associated with running a Pod. Declaring overhead allows the cluster (including the scheduler) to account for it when making decisions about Pods and resources.

To use Pod overhead, you must have the PodOverhead [feature gate](#) enabled (it is on by default).

Pod overhead is defined in RuntimeClass through the `overhead` fields. Through the use of these fields, you can specify the overhead of running pods utilizing this RuntimeClass and ensure these overheads are accounted for in Kubernetes.

What's next

- [RuntimeClass Design](#)
- [RuntimeClass Scheduling Design](#)
- Read about the [Pod Overhead](#) concept
- [PodOverhead Feature Design](#)

Feedback

Was this page helpful?

Yes No

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

Last modified October 26, 2020 at 3:27 PM PST: [Update content/en/docs/concepts/containers/runtime-class.md \(ca7cb78ca\)](#)

[Edit this page](#) [Create child page](#) [Create an issue](#)

- [Motivation](#)
- [Setup](#)
 - [1. Configure the CRI implementation on nodes](#)
 - [2. Create the corresponding RuntimeClass resources](#)
- [Usage](#)
 - [CRI Configuration](#)
- [Scheduling](#)
 - [Pod Overhead](#)
- [What's next](#)

Container Lifecycle Hooks

This page describes how kubelet managed Containers can use the Container lifecycle hook framework to run code triggered by events during their management lifecycle.

Overview

Analogous to many programming language frameworks that have component lifecycle hooks, such as Angular, Kubernetes provides Containers with lifecycle hooks. The hooks enable Containers to be aware of events in their management lifecycle and run code implemented in a handler when the corresponding lifecycle hook is executed.

Container hooks

There are two hooks that are exposed to Containers:

`PostStart`

This hook is executed immediately after a container is created. However, there is no guarantee that the hook will execute before the container ENTRYPPOINT. No parameters are passed to the handler.

`PreStop`

This hook is called immediately before a container is terminated due to an API request or management event such as liveness probe failure, preemption, resource contention and others. A call to the preStop hook fails if the container is already in terminated or completed state. It is blocking, meaning it is synchronous, so it must complete before the signal to stop the container can be sent. No parameters are passed to the handler.

A more detailed description of the termination behavior can be found in [Termination of Pods](#).

Hook handler implementations

Containers can access a hook by implementing and registering a handler for that hook. There are two types of hook handlers that can be implemented for Containers:

- *Exec - Executes a specific command, such as `pre-stop.sh`, inside the cgroups and namespaces of the Container. Resources consumed by the command are counted against the Container.*
- *HTTP - Executes an HTTP request against a specific endpoint on the Container.*

Hook handler execution

When a Container lifecycle management hook is called, the Kubernetes management system execute the handler according to the hook action, exec and tcpSocket are executed in the container, and httpGet is executed by the kubelet process.

Hook handler calls are synchronous within the context of the Pod containing the Container. This means that for a PostStart hook, the Container ENTRYPOINT and hook fire asynchronously. However, if the hook takes too long to run or hangs, the Container cannot reach a running state.

PreStop hooks are not executed asynchronously from the signal to stop the Container; the hook must complete its execution before the signal can be sent. If a PreStop hook hangs during execution, the Pod's phase will be Terminating and remain there until the Pod is killed after its terminationGracePeriodSeconds expires. This grace period applies to the total time it takes for both the PreStop hook to execute and for the Container to stop normally. If, for example, terminationGracePeriodSeconds is 60, and the hook takes 55 seconds to complete, and the Container takes 10 seconds to stop normally after receiving the signal, then the Container will be killed before it can stop normally, since terminationGracePeriodSeconds is less than the total time (55+10) it takes for these two things to happen.

If either a PostStart or PreStop hook fails, it kills the Container.

Users should make their hook handlers as lightweight as possible. There are cases, however, when long running commands make sense, such as when saving state prior to stopping a Container.

Hook delivery guarantees

Hook delivery is intended to be at least once, which means that a hook may be called multiple times for any given event, such as for PostStart or PreStop. It is up to the hook implementation to handle this correctly.

Generally, only single deliveries are made. If, for example, an HTTP hook receiver is down and is unable to take traffic, there is no attempt to resend. In some rare cases, however, double delivery may occur. For instance, if a kubelet restarts in the middle of sending a hook, the hook might be resent after the kubelet comes back up.

Debugging Hook handlers

The logs for a Hook handler are not exposed in Pod events. If a handler fails for some reason, it broadcasts an event. For PostStart, this is the FailedPostStartHook event, and for PreStop, this is the FailedPreStopHook event. You can see these events by running kubectl describe pod <pod_name>. Here is some example output of events from running this command:

Events:

FirstSeen	LastSeen	Count
-----------	----------	-------

<i>From</i>		<i>Type</i>	<i>Reason</i>	<i>Message</i>
	<i>SubObjectPath</i>			
-----	-----	-----	-----	-----
-----	-----	-----	-----	-----
1m	1m	1	{default-scheduler }	
Normal	Scheduled			Successfully assigned test-1730497541-cq1d2 to gke-test-cluster-default-pool-a07e5d30-siqd
1m	1m	1	{kubelet gke-test-cluster-default-pool-a07e5d30-siqd} spec.containers{main}	Normal
Pulling				pulling image "test:1.0"
1m	1m	1	{kubelet gke-test-cluster-default-pool-a07e5d30-siqd} spec.containers{main}	Normal
Created				Created container with docker id 5c6a256a2567; Security:[seccomp=unconfined]
1m	1m	1	{kubelet gke-test-cluster-default-pool-a07e5d30-siqd} spec.containers{main}	Normal
Pulled				Successfully pulled image "test:1.0"
1m	1m	1	{kubelet gke-test-cluster-default-pool-a07e5d30-siqd} spec.containers{main}	Normal
Started				Started container with docker id 5c6a256a2567
38s	38s	1	{kubelet gke-test-cluster-default-pool-a07e5d30-siqd} spec.containers{main}	Normal
Killing				Killing container with docker id 5c6a256a2567: PostStart handler: Error executing in Docker Container: 1
37s	37s	1	{kubelet gke-test-cluster-default-pool-a07e5d30-siqd} spec.containers{main}	Normal
Killing				Killing container with docker id 8df9fdfd7054: PostStart handler: Error executing in Docker Container: 1
38s	37s	2	{kubelet gke-test-cluster-default-pool-a07e5d30-siqd}	Warning
FailedSync				Error syncing pod, skipping: failed to "StartContainer" for "main" with RunContainerError: "PostStart handler: Error executing in Docker Container: 1"
1m	22s	2	{kubelet gke-test-cluster-default-pool-a07e5d30-siqd} spec.containers{main}	Warning
FailedPostStartHook				

What's next

- Learn more about the [Container environment](#).
- Get hands-on experience [attaching handlers to Container lifecycle events](#).

Feedback

Was this page helpful?

Yes No

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

Last modified September 05, 2020 at 7:58 PM PST: [clarify the execution contexts of hook handlers with different action types are different](#) ([10ebb11f2](#))

[Edit this page](#) [Create child page](#) [Create an issue](#)

- [Overview](#)
- [Container hooks](#)
 - [Hook handler implementations](#)
 - [Hook handler execution](#)
 - [Hook delivery guarantees](#)
 - [Debugging Hook handlers](#)
- [What's next](#)

Workloads

Understand Pods, the smallest deployable compute object in Kubernetes, and the higher-level abstractions that help you to run them.

A workload is an application running on Kubernetes. Whether your workload is a single component or several that work together, on Kubernetes you run it inside a set of [pods](#). In Kubernetes, a Pod represents a set of running [containers](#) on your cluster.

Kubernetes pods have a [defined lifecycle](#). For example, once a pod is running in your cluster then a critical fault on the [node](#) where that pod is running means that all the pods on that node fail. Kubernetes treats that level of failure as final: you would need to create a new Pod to recover, even if the node later becomes healthy.

However, to make life considerably easier, you don't need to manage each Pod directly. Instead, you can use workload resources that manage a set of pods on your behalf. These resources configure [controllers](#) that make sure the right number of the right kind of pod are running, to match the state you specified.

Kubernetes provides several built-in workload resources:

- [Deployment](#) and [ReplicaSet](#) (replacing the legacy resource [ReplicationController](#)). Deployment is a good fit for managing a stateless application workload on your cluster, where any Pod in the Deployment is interchangeable and can be replaced if needed.

- [StatefulSet](#) lets you run one or more related Pods that do track state somehow. For example, if your workload records data persistently, you can run a StatefulSet that matches each Pod with a [PersistentVolume](#). Your code, running in the Pods for that StatefulSet, can replicate data to other Pods in the same StatefulSet to improve overall resilience.
- [DaemonSet](#) defines Pods that provide node-local facilities. These might be fundamental to the operation of your cluster, such as a networking helper tool, or be part of an [add-on](#).
Every time you add a node to your cluster that matches the specification in a DaemonSet, the control plane schedules a Pod for that DaemonSet onto the new node.
- [Job](#) and [CronJob](#) define tasks that run to completion and then stop. Jobs represent one-off tasks, whereas CronJobs recur according to a schedule.

In the wider Kubernetes ecosystem, you can find third-party workload resources that provide additional behaviors. Using a [custom resource definition](#), you can add in a third-party workload resource if you want a specific behavior that's not part of Kubernetes' core. For example, if you wanted to run a group of Pods for your application but stop work unless all the Pods are available (perhaps for some high-throughput distributed task), then you can implement or install an extension that does provide that feature.

What's next

As well as reading about each resource, you can learn about specific tasks that relate to them:

- [Run a stateless application using a Deployment](#)
- Run a stateful application either as a [single instance](#) or as a [replicated set](#)
- [Run automated tasks with a CronJob](#)

To learn about Kubernetes' mechanisms for separating code from configuration, visit [Configuration](#).

There are two supporting concepts that provide backgrounds about how Kubernetes manages pods for applications:

- [Garbage collection](#) tidies up objects from your cluster after their owning resource has been removed.
- The [time-to-live after finished controller](#) removes Jobs once a defined time has passed since they completed.

Once your application is running, you might want to make it available on the internet as a [Service](#) or, for web application only, using an [Ingress](#).

Pods

Pods are the smallest deployable units of computing that you can create and manage in Kubernetes.

A Pod (as in a pod of whales or pea pod) is a group of one or more [containers](#), with shared storage/network resources, and a specification for how to run the containers. A Pod's contents are always co-located and co-scheduled, and run in a shared context. A Pod models an application-specific "logical host": it contains one or more application containers which are relatively tightly coupled. In non-cloud contexts, applications executed on the same physical or virtual machine are analogous to cloud applications executed on the same logical host.

As well as application containers, a Pod can contain [init containers](#) that run during Pod startup. You can also inject [ephemeral containers](#) for debugging if your cluster offers this.

What is a Pod?

Note: While Kubernetes supports more [container runtimes](#) than just Docker, [Docker](#) is the most commonly known runtime, and it helps to describe Pods using some terminology from Docker.

The shared context of a Pod is a set of Linux namespaces, cgroups, and potentially other facets of isolation - the same things that isolate a Docker container. Within a Pod's context, the individual applications may have further sub-isolations applied.

In terms of Docker concepts, a Pod is similar to a group of Docker containers with shared namespaces and shared filesystem volumes.

Using Pods

Usually you don't need to create Pods directly, even singleton Pods. Instead, create them using workload resources such as [Deployment](#) or [Job](#). If your Pods need to track state, consider the [StatefulSet](#) resource.

Pods in a Kubernetes cluster are used in two main ways:

- **Pods that run a single container.** The "one-container-per-Pod" model is the most common Kubernetes use case; in this case, you can think of a Pod as a wrapper around a single container; Kubernetes manages Pods rather than managing the containers directly.
- **Pods that run multiple containers that need to work together.** A Pod can encapsulate an application composed of multiple co-located containers that are tightly coupled and need to share resources. These co-located containers form a single cohesive unit of serviceâ€”for example, one container serving data stored in a shared volume to the

public, while a separate sidecar container refreshes or updates those files. The Pod wraps these containers, storage resources, and an ephemeral network identity together as a single unit.

Note: Grouping multiple co-located and co-managed containers in a single Pod is a relatively advanced use case. You should use this pattern only in specific instances in which your containers are tightly coupled.

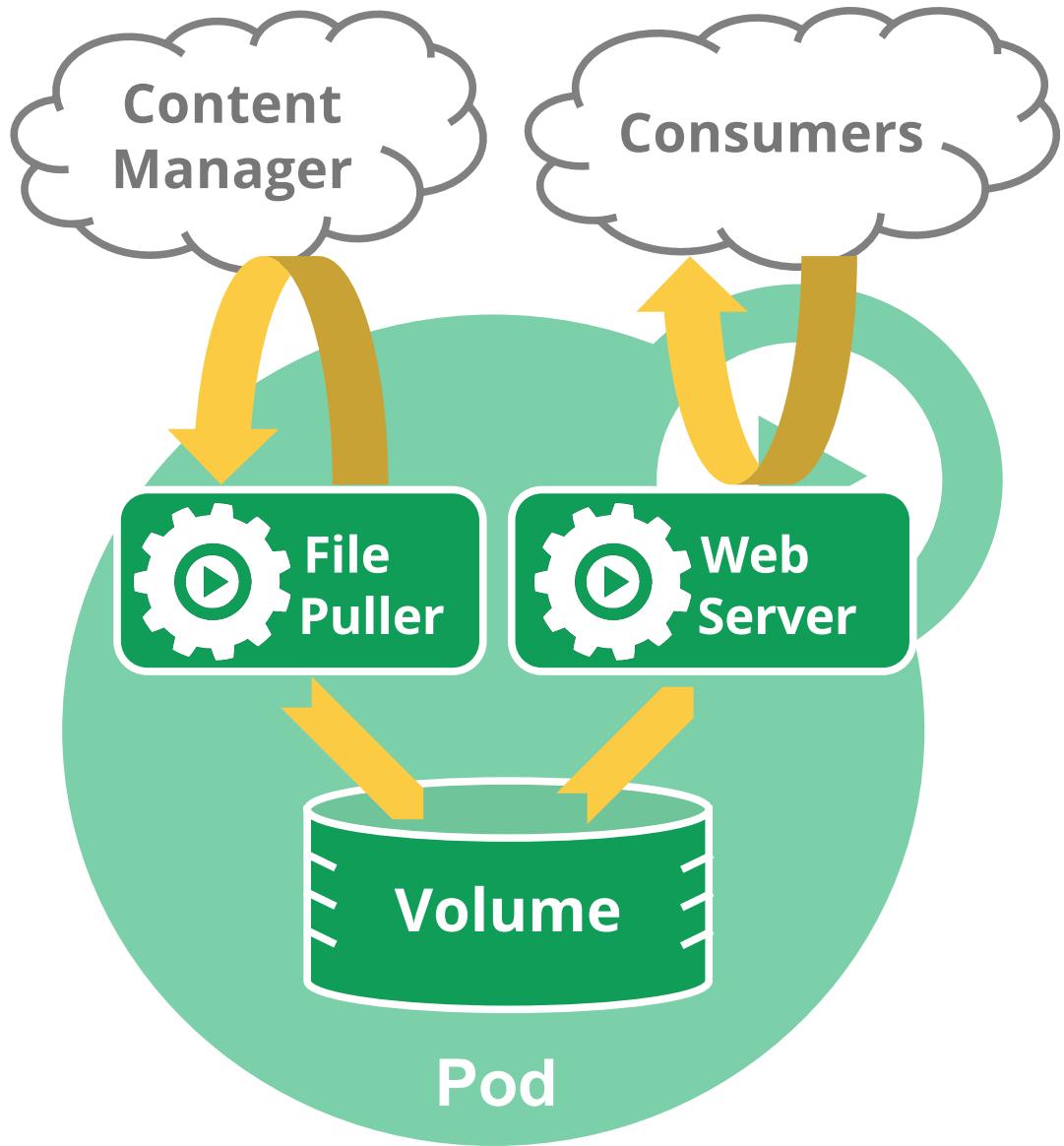
Each Pod is meant to run a single instance of a given application. If you want to scale your application horizontally (to provide more overall resources by running more instances), you should use multiple Pods, one for each instance. In Kubernetes, this is typically referred to as replication. Replicated Pods are usually created and managed as a group by a workload resource and its [controller](#).

See [Pods and controllers](#) for more information on how Kubernetes uses workload resources, and their controllers, to implement application scaling and auto-healing.

How Pods manage multiple containers

Pods are designed to support multiple cooperating processes (as containers) that form a cohesive unit of service. The containers in a Pod are automatically co-located and co-scheduled on the same physical or virtual machine in the cluster. The containers can share resources and dependencies, communicate with one another, and coordinate when and how they are terminated.

For example, you might have a container that acts as a web server for files in a shared volume, and a separate "sidecar" container that updates those files from a remote source, as in the following diagram:



Some Pods have [init containers](#) as well as [app containers](#). Init containers run and complete before the app containers are started.

Pods natively provide two kinds of shared resources for their constituent containers: [networking](#) and [storage](#).

Working with Pods

You'll rarely create individual Pods directly in Kubernetes—“even singleton Pods. This is because Pods are designed as relatively ephemeral, disposable entities. When a Pod gets created (directly by you, or indirectly by a [controller](#)), the new Pod is scheduled to run on a [Node](#) in your cluster. The Pod remains on that node until the Pod finishes execution, the Pod object is deleted, the Pod is evicted for lack of resources, or the node fails.

Note: Restarting a container in a Pod should not be confused with restarting a Pod. A Pod is not a process, but an environment for running container(s). A Pod persists until it is deleted.

When you create the manifest for a Pod object, make sure the name specified is a valid [DNS subdomain name](#).

Pods and controllers

You can use workload resources to create and manage multiple Pods for you. A controller for the resource handles replication and rollout and automatic healing in case of Pod failure. For example, if a Node fails, a controller notices that Pods on that Node have stopped working and creates a replacement Pod. The scheduler places the replacement Pod onto a healthy Node.

Here are some examples of workload resources that manage one or more Pods:

- [Deployment](#)
- [StatefulSet](#)
- [DaemonSet](#)

Pod templates

Controllers for [workload](#) resources create Pods from a pod template and manage those Pods on your behalf.

PodTemplates are specifications for creating Pods, and are included in workload resources such as [Deployments](#), [Jobs](#), and [DaemonSets](#).

Each controller for a workload resource uses the PodTemplate inside the workload object to make actual Pods. The PodTemplate is part of the desired state of whatever workload resource you used to run your app.

The sample below is a manifest for a simple Job with a template that starts one container. The container in that Pod prints a message then pauses.

```
apiVersion: batch/v1
kind: Job
metadata:
  name: hello
spec:
  template:
    # This is the pod template
    spec:
      containers:
        - name: hello
          image: busybox
          command: ['sh', '-c', 'echo "Hello, Kubernetes!" && sleep 3600']
  restartPolicy: OnFailure
  # The pod template ends here
```

Modifying the pod template or switching to a new pod template has no direct effect on the Pods that already exist. If you change the pod template

for a workload resource, that resource needs to create replacement Pods that use the updated template.

For example, the `StatefulSet` controller ensures that the running Pods match the current pod template for each `StatefulSet` object. If you edit the `StatefulSet` to change its pod template, the `StatefulSet` starts to create new Pods based on the updated template. Eventually, all of the old Pods are replaced with new Pods, and the update is complete.

Each workload resource implements its own rules for handling changes to the Pod template. If you want to read more about `StatefulSet` specifically, read [Update strategy](#) in the `StatefulSet` Basics tutorial.

On Nodes, the [kubelet](#) does not directly observe or manage any of the details around pod templates and updates; those details are abstracted away. That abstraction and separation of concerns simplifies system semantics, and makes it feasible to extend the cluster's behavior without changing existing code.

Resource sharing and communication

Pods enable data sharing and communication among their constituent containers.

Storage in Pods

A Pod can specify a set of shared storage [volumes](#). All containers in the Pod can access the shared volumes, allowing those containers to share data. Volumes also allow persistent data in a Pod to survive in case one of the containers within needs to be restarted. See [Storage](#) for more information on how Kubernetes implements shared storage and makes it available to Pods.

Pod networking

Each Pod is assigned a unique IP address for each address family. Every container in a Pod shares the network namespace, including the IP address and network ports. Inside a Pod (and **only** then), the containers that belong to the Pod can communicate with one another using `localhost`. When containers in a Pod communicate with entities outside the Pod, they must coordinate how they use the shared network resources (such as ports). Within a Pod, containers share an IP address and port space, and can find each other via `localhost`. The containers in a Pod can also communicate with each other using standard inter-process communications like SystemV semaphores or POSIX shared memory. Containers in different Pods have distinct IP addresses and can not communicate by IPC without [special configuration](#). Containers that want to interact with a container running in a different Pod can use IP networking to communicate.

Containers within the Pod see the system hostname as being the same as the configured name for the Pod. There's more about this in the [networking](#) section.

Privileged mode for containers

Any container in a Pod can enable privileged mode, using the `privileged` flag on the [security context](#) of the container spec. This is useful for containers that want to use operating system administrative capabilities such as manipulating the network stack or accessing hardware devices. Processes within a privileged container get almost the same privileges that are available to processes outside a container.

Note: Your [container runtime](#) must support the concept of a privileged container for this setting to be relevant.

Static Pods

Static Pods are managed directly by the kubelet daemon on a specific node, without the [API server](#) observing them. Whereas most Pods are managed by the control plane (for example, a [Deployment](#)), for static Pods, the kubelet directly supervises each static Pod (and restarts it if it fails).

Static Pods are always bound to one [Kubelet](#) on a specific node. The main use for static Pods is to run a self-hosted control plane: in other words, using the kubelet to supervise the individual [control plane components](#).

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.

What's next

- Learn about the [lifecycle of a Pod](#).
- Learn about [RuntimeClass](#) and how you can use it to configure different Pods with different container runtime configurations.
- Read about [Pod topology spread constraints](#).
- Read about [PodDisruptionBudget](#) and how you can use it to manage application availability during disruptions.
- Pod is a top-level resource in the Kubernetes REST API. The [Pod](#) object definition describes the object in detail.
- [The Distributed System Toolkit: Patterns for Composite Containers](#) explains common layouts for Pods with more than one container.

To understand the context for why Kubernetes wraps a common Pod API in other resources (such as [StatefulSets](#) or [Deployments](#)), you can read about the prior art, including:

- [Aurora](#)
- [Borg](#)

- [Marathon](#)
- [Omega](#)
- [Tupperware](#).

Pod Lifecycle

This page describes the lifecycle of a Pod. Pods follow a defined lifecycle, starting in the Pending [phase](#), moving through Running if at least one of its primary containers starts OK, and then through either the Succeeded or Failed phases depending on whether any container in the Pod terminated in failure.

Whilst a Pod is running, the kubelet is able to restart containers to handle some kind of faults. Within a Pod, Kubernetes tracks different container [states](#) and determines what action to take to make the Pod healthy again.

In the Kubernetes API, Pods have both a specification and an actual status. The status for a Pod object consists of a set of [Pod conditions](#). You can also inject [custom readiness information](#) into the condition data for a Pod, if that is useful to your application.

Pods are only [scheduled](#) once in their lifetime. Once a Pod is scheduled (assigned) to a Node, the Pod runs on that Node until it stops or is [terminated](#).

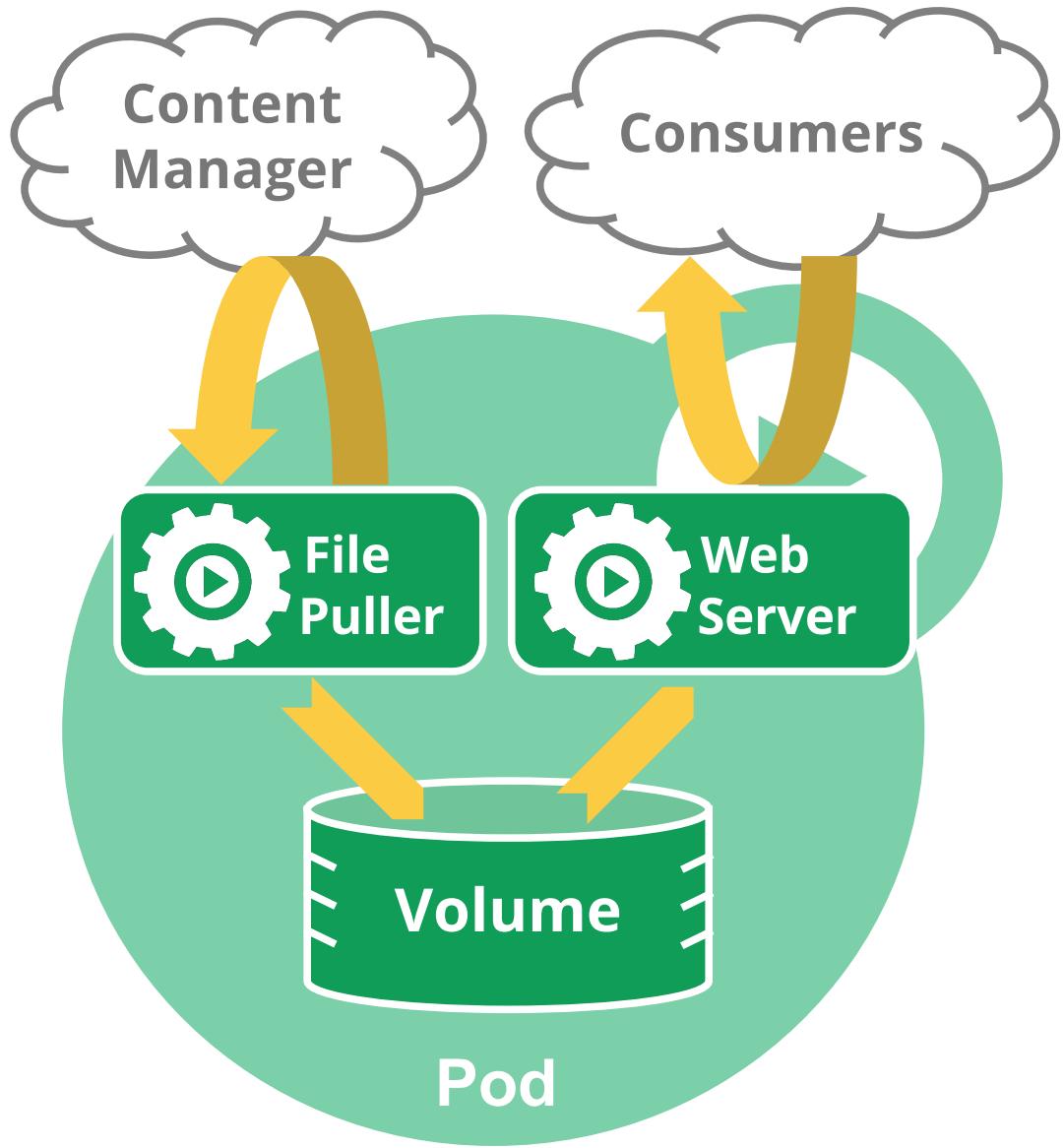
Pod lifetime

Like individual application containers, Pods are considered to be relatively ephemeral (rather than durable) entities. Pods are created, assigned a unique ID ([UID](#)), and scheduled to nodes where they remain until termination (according to restart policy) or deletion. If a [Node](#) dies, the Pods scheduled to that node are [scheduled for deletion](#) after a timeout period.

Pods do not, by themselves, self-heal. If a Pod is scheduled to a [node](#) that then fails, or if the scheduling operation itself fails, the Pod is deleted; likewise, a Pod won't survive an eviction due to a lack of resources or Node maintenance. Kubernetes uses a higher-level abstraction, called a [controller](#), that handles the work of managing the relatively disposable Pod instances.

A given Pod (as defined by a UID) is never "rescheduled" to a different node; instead, that Pod can be replaced by a new, near-identical Pod, with even the same name if desired, but with a different UID.

When something is said to have the same lifetime as a Pod, such as a [volume](#), that means that the thing exists as long as that specific Pod (with that exact UID) exists. If that Pod is deleted for any reason, and even if an identical replacement is created, the related thing (a volume, in this example) is also destroyed and created anew.



Pod diagram

A multi-container Pod that contains a file puller and a web server that uses a persistent volume for shared storage between the containers.

Pod phase

A Pod's *status* field is a [PodStatus](#) object, which has a *phase* field.

The phase of a Pod is a simple, high-level summary of where the Pod is in its lifecycle. The phase is not intended to be a comprehensive rollup of observations of container or Pod state, nor is it intended to be a comprehensive state machine.

The number and meanings of Pod phase values are tightly guarded. Other than what is documented here, nothing should be assumed about Pods that have a given phase value.

Here are the possible values for phase:

Value	Description
Pending	The Pod has been accepted by the Kubernetes cluster, but one or more of the containers has not been set up and made ready to run. This includes time a Pod spends waiting to be scheduled as well as the time spent downloading container images over the network.
Running	The Pod has been bound to a node, and all of the containers have been created. At least one container is still running, or is in the process of starting or restarting.
Succeeded	All containers in the Pod have terminated in success, and will not be restarted.
Failed	All containers in the Pod have terminated, and at least one container has terminated in failure. That is, the container either exited with non-zero status or was terminated by the system.
Unknown	For some reason the state of the Pod could not be obtained. This phase typically occurs due to an error in communicating with the node where the Pod should be running.

If a node dies or is disconnected from the rest of the cluster, Kubernetes applies a policy for setting the phase of all Pods on the lost node to Failed.

Container states

As well as the [phase](#) of the Pod overall, Kubernetes tracks the state of each container inside a Pod. You can use [container lifecycle hooks](#) to trigger events to run at certain points in a container's lifecycle.

Once the [scheduler](#) assigns a Pod to a Node, the kubelet starts creating containers for that Pod using a [container runtime](#). There are three possible container states: Waiting, Running, and Terminated.

To check the state of a Pod's containers, you can use `kubectl describe pod <name-of-pod>`. The output shows the state for each container within that Pod.

Each state has a specific meaning:

Waiting

If a container is not in either the Running or Terminated state, it is Waiting. A container in the Waiting state is still running the operations it requires in order to complete start up: for example, pulling the container image from a container image registry, or applying [Secret](#) data. When you use `kubectl` to query a Pod with a container that is Waiting, you also see a Reason field to summarize why the container is in that state.

Running

The Running status indicates that a container is executing without issues. If there was a postStart hook configured, it has already executed and finished. When you use kubectl to query a Pod with a container that is Running, you also see information about when the container entered the Running state.

Terminated

A container in the Terminated state began execution and then either ran to completion or failed for some reason. When you use kubectl to query a Pod with a container that is Terminated, you see a reason, an exit code, and the start and finish time for that container's period of execution.

If a container has a preStop hook configured, that runs before the container enters the Terminated state.

Container restart policy

The spec of a Pod has a restartPolicy field with possible values Always, OnFailure, and Never. The default value is Always.

The restartPolicy applies to all containers in the Pod. restartPolicy only refers to restarts of the containers by the kubelet on the same node. After containers in a Pod exit, the kubelet restarts them with an exponential back-off delay (10s, 20s, 40s, ...), that is capped at five minutes. Once a container has executed for 10 minutes without any problems, the kubelet resets the restart backoff timer for that container.

Pod conditions

A Pod has a PodStatus, which has an array of [PodConditions](#) through which the Pod has or has not passed:

- *PodScheduled: the Pod has been scheduled to a node.*
- *ContainersReady: all containers in the Pod are ready.*
- *Initialized: all [init containers](#) have started successfully.*
- *Ready: the Pod is able to serve requests and should be added to the load balancing pools of all matching Services.*

Field name	Description
type	Name of this Pod condition.
status	Indicates whether that condition is applicable, with possible values "True", "False", or "Unknown".
lastProbeTime	Timestamp of when the Pod condition was last probed.
lastTransitionTime	Timestamp for when the Pod last transitioned from one status to another.
reason	Machine-readable, UpperCamelCase text indicating the reason for the condition's last transition.

Field name	Description
message	Human-readable message indicating details about the last status transition.

Pod readiness

FEATURE STATE: Kubernetes v1.14 [stable]

Your application can inject extra feedback or signals into `PodStatus: Pod` readiness. To use this, set `readinessGates` in the Pod's spec to specify a list of additional conditions that the kubelet evaluates for Pod readiness.

Readiness gates are determined by the current state of `status.condition` fields for the Pod. If Kubernetes cannot find such a condition in the `status.conditions` field of a Pod, the status of the condition is defaulted to "False".

Here is an example:

```
kind: Pod
...
spec:
  readinessGates:
    - conditionType: "www.example.com/feature-1"
status:
  conditions:
    - type: Ready # a built in
      PodCondition
        status: "False"
        lastProbeTime: null
        lastTransitionTime: 2018-01-01T00:00:00Z
    - type: "www.example.com/feature-1" # an extra
      PodCondition
        status: "False"
        lastProbeTime: null
        lastTransitionTime: 2018-01-01T00:00:00Z
  containerStatuses:
    - containerID: docker://abcd...
      ready: true
...

```

The Pod conditions you add must have names that meet the Kubernetes [label key format](#).

Status for Pod readiness

The `kubectl patch` command does not support patching object status. To set these `status.conditions` for the pod, applications and [operators](#) should use the `PATCH` action. You can use a [Kubernetes client library](#) to write code that sets custom Pod conditions for Pod readiness.

For a Pod that uses custom conditions, that Pod is evaluated to be ready only when both the following statements apply:

- All containers in the Pod are ready.
- All conditions specified in `readinessGates` are True.

When a Pod's containers are Ready but at least one custom condition is missing or False, the kubelet sets the Pod's [condition](#) to `ContainersReady`.

Container probes

A [Probe](#) is a diagnostic performed periodically by the [kubelet](#) on a Container. To perform a diagnostic, the kubelet calls a [Handler](#) implemented by the container. There are three types of handlers:

- [`ExecAction`](#): Executes a specified command inside the container. The diagnostic is considered successful if the command exits with a status code of 0.
- [`TCPSocketAction`](#): Performs a TCP check against the Pod's IP address on a specified port. The diagnostic is considered successful if the port is open.
- [`HTTPGetAction`](#): Performs an HTTP GET request against the Pod's IP address on a specified port and path. The diagnostic is considered successful if the response has a status code greater than or equal to 200 and less than 400.

Each probe has one of three results:

- *Success*: The container passed the diagnostic.
- *Failure*: The container failed the diagnostic.
- *Unknown*: The diagnostic failed, so no action should be taken.

The kubelet can optionally perform and react to three kinds of probes on running containers:

- [`livenessProbe`](#): Indicates whether the container is running. If the liveness probe fails, the kubelet kills the container, and the container is subjected to its [restart policy](#). If a Container does not provide a liveness probe, the default state is Success.
- [`readinessProbe`](#): Indicates whether the container is ready to respond to requests. If the readiness probe fails, the endpoints controller removes the Pod's IP address from the endpoints of all Services that match the Pod. The default state of readiness before the initial delay is Failure. If a Container does not provide a readiness probe, the default state is Success.
- [`startupProbe`](#): Indicates whether the application within the container is started. All other probes are disabled if a startup probe is provided, until it succeeds. If the startup probe fails, the kubelet kills the container, and the container is subjected to its [restart policy](#). If a

Container does not provide a startup probe, the default state is Success.

For more information about how to set up a liveness, readiness, or startup probe, see [Configure Liveness, Readiness and Startup Probes](#).

When should you use a liveness probe?

FEATURE STATE: Kubernetes v1.0 [stable]

If the process in your container is able to crash on its own whenever it encounters an issue or becomes unhealthy, you do not necessarily need a liveness probe; the kubelet will automatically perform the correct action in accordance with the Pod's restartPolicy.

If you'd like your container to be killed and restarted if a probe fails, then specify a liveness probe, and specify a restartPolicy of Always or OnFailure.

When should you use a readiness probe?

FEATURE STATE: Kubernetes v1.0 [stable]

If you'd like to start sending traffic to a Pod only when a probe succeeds, specify a readiness probe. In this case, the readiness probe might be the same as the liveness probe, but the existence of the readiness probe in the spec means that the Pod will start without receiving any traffic and only start receiving traffic after the probe starts succeeding. If your container needs to work on loading large data, configuration files, or migrations during startup, specify a readiness probe.

If you want your container to be able to take itself down for maintenance, you can specify a readiness probe that checks an endpoint specific to readiness that is different from the liveness probe.

Note: *If you just want to be able to drain requests when the Pod is deleted, you do not necessarily need a readiness probe; on deletion, the Pod automatically puts itself into an unready state regardless of whether the readiness probe exists. The Pod remains in the unready state while it waits for the containers in the Pod to stop.*

When should you use a startup probe?

FEATURE STATE: Kubernetes v1.20 [stable]

Startup probes are useful for Pods that have containers that take a long time to come into service. Rather than set a long liveness interval, you can configure a separate configuration for probing the container as it starts up, allowing a time longer than the liveness interval would allow.

If your container usually starts in more than `initialDelaySeconds` + `failureThreshold` – `periodSeconds`, you should specify a startup probe that checks the same endpoint as the liveness probe. The default for `periodSeconds` is 30s. You should then set its `failureThreshold` high enough to allow the container to start, without changing the default values of the liveness probe. This helps to protect against deadlocks.

Termination of Pods

Because Pods represent processes running on nodes in the cluster, it is important to allow those processes to gracefully terminate when they are no longer needed (rather than being abruptly stopped with a `KILL` signal and having no chance to clean up).

The design aim is for you to be able to request deletion and know when processes terminate, but also be able to ensure that deletes eventually complete. When you request deletion of a Pod, the cluster records and tracks the intended grace period before the Pod is allowed to be forcefully killed. With that forceful shutdown tracking in place, the [kubelet](#) attempts graceful shutdown.

Typically, the container runtime sends a `TERM` signal to the main process in each container. Many container runtimes respect the `STOPSIGAL` value defined in the container image and send this instead of `TERM`. Once the grace period has expired, the `KILL` signal is sent to any remaining processes, and the Pod is then deleted from the [API Server](#). If the kubelet or the container runtime's management service is restarted while waiting for processes to terminate, the cluster retries from the start including the full original grace period.

An example flow:

1. You use the `kubectl` tool to manually delete a specific Pod, with the default grace period (30 seconds).
2. The Pod in the API server is updated with the time beyond which the Pod is considered "dead" along with the grace period. If you use `kubectl describe` to check on the Pod you're deleting, that Pod shows up as "Terminating". On the node where the Pod is running: as soon as the kubelet sees that a Pod has been marked as terminating (a graceful shutdown duration has been set), the kubelet begins the local Pod shutdown process.
 1. If one of the Pod's containers has defined a [preStop hook](#), the kubelet runs that hook inside of the container. If the `preStop` hook is still running after the grace period expires, the kubelet requests a small, one-off grace period extension of 2 seconds.

Note: If the `preStop` hook needs longer to complete than the default grace period allows, you must modify `terminationGracePeriodSeconds` to suit this.

2. The kubelet triggers the container runtime to send a `TERM` signal to process 1 inside each container.

Note: The containers in the Pod receive the `TERM` signal at different times and in an arbitrary order. If the order of shutdowns matters, consider using a `preStop` hook to synchronize.

3. At the same time as the kubelet is starting graceful shutdown, the control plane removes that shutting-down Pod from Endpoints (and, if enabled, `EndpointSlice`) objects where these represent a [Service](#) with a configured [selector](#). [ReplicaSets](#) and other workload resources no longer treat the shutting-down Pod as a valid, in-service replica. Pods that shut down slowly cannot continue to serve traffic as load balancers (like the service proxy) remove the Pod from the list of endpoints as soon as the termination grace period begins.
4. When the grace period expires, the kubelet triggers forcible shutdown. The container runtime sends `SIGKILL` to any processes still running in any container in the Pod. The kubelet also cleans up a hidden `pause` container if that container runtime uses one.
5. The kubelet triggers forcible removal of Pod object from the API server, by setting grace period to 0 (immediate deletion).
6. The API server deletes the Pod's API object, which is then no longer visible from any client.

Forced Pod termination

Caution: Forced deletions can be potentially disruptive for some workloads and their Pods.

By default, all deletes are graceful within 30 seconds. The `kubectl delete` command supports the `--grace-period=<seconds>` option which allows you to override the default and specify your own value.

Setting the grace period to 0 forcibly and immediately deletes the Pod from the API server. If the pod was still running on a node, that forcible deletion triggers the kubelet to begin immediate cleanup.

Note: You must specify an additional flag `--force` along with `--grace-period=0` in order to perform force deletions.

When a force deletion is performed, the API server does not wait for confirmation from the kubelet that the Pod has been terminated on the node it was running on. It removes the Pod in the API immediately so a new Pod can be created with the same name. On the node, Pods that are set to terminate immediately will still be given a small grace period before being force killed.

If you need to force-delete Pods that are part of a `StatefulSet`, refer to the task documentation for [deleting Pods from a StatefulSet](#).

Garbage collection of failed Pods

For failed Pods, the API objects remain in the cluster's API until a human or controller process explicitly removes them.

The control plane cleans up terminated Pods (with a phase of `Succeeded` or `Failed`), when the number of Pods exceeds the configured threshold (determined by `terminated-pod-gc-threshold` in the `kube-controller-manager`). This avoids a resource leak as Pods are created and terminated over time.

What's next

- Get hands-on experience [attaching handlers to Container lifecycle events](#).
- Get hands-on experience [configuring Liveness, Readiness and Startup Probes](#).
- Learn more about [container lifecycle hooks](#).
- For detailed information about Pod / Container status in the API, see [PodStatus](#) and [ContainerStatus](#).

Feedback

Was this page helpful?

Yes No

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

Last modified November 17, 2020 at 8:11 PM PST: [Fix typo in container restart policy #25041 \(983df6134\)](#)

[Edit this page](#) [Create child page](#) [Create an issue](#)

- [Pod lifetime](#)
- [Pod phase](#)
- [Container states](#)
 - [Waiting](#)
 - [Running](#)
 - [Terminated](#)
- [Container restart policy](#)
- [Pod conditions](#)
 - [Pod readiness](#)
 - [Status for Pod readiness](#)
- [Container probes](#)
 - [When should you use a liveness probe?](#)
 - [When should you use a readiness probe?](#)
 - [When should you use a startup probe?](#)
- [Termination of Pods](#)
 - [Forced Pod termination](#)
 - [Garbage collection of failed Pods](#)
- [What's next](#)

Init Containers

This page provides an overview of init containers: specialized containers that run before app containers in a [Pod](#). Init containers can contain utilities or setup scripts not present in an app image.

You can specify init containers in the Pod specification alongside the `containers` array (which describes app containers).

Understanding init containers

A [Pod](#) can have multiple containers running apps within it, but it can also have one or more init containers, which are run before the app containers are started.

Init containers are exactly like regular containers, except:

- Init containers always run to completion.
- Each init container must complete successfully before the next one starts.

If a Pod's init container fails, the kubelet repeatedly restarts that init container until it succeeds. However, if the Pod has a `restartPolicy` of `Never`, and an init container fails during startup of that Pod, Kubernetes treats the overall Pod as failed.

To specify an init container for a Pod, add the `initContainers` field into the Pod specification, as an array of objects of type [Container](#), alongside the app containers array. The status of the init containers is returned in `.status.initContainerStatuses` field as an array of the container statuses (similar to the `.status.containerStatuses` field).

Differences from regular containers

Init containers support all the fields and features of app containers, including resource limits, volumes, and security settings. However, the resource requests and limits for an init container are handled differently, as documented in [Resources](#).

Also, init containers do not support `lifecycle`, `livenessProbe`, `readinessProbe`, or `startupProbe` because they must run to completion before the Pod can be ready.

If you specify multiple init containers for a Pod, kubelet runs each init container sequentially. Each init container must succeed before the next can run. When all of the init containers have run to completion, kubelet initializes the application containers for the Pod and runs them as usual.

Using init containers

Because init containers have separate images from app containers, they have some advantages for start-up related code:

- Init containers can contain utilities or custom code for setup that are not present in an app image. For example, there is no need to make an image *FROM* another image just to use a tool like `sed`, `awk`, `python`, or `dig` during setup.
- The application image builder and deployer roles can work independently without the need to jointly build a single app image.
- Init containers can run with a different view of the filesystem than app containers in the same Pod. Consequently, they can be given access to [Secrets](#) that app containers cannot access.
- Because init containers run to completion before any app containers start, init containers offer a mechanism to block or delay app container startup until a set of preconditions are met. Once preconditions are met, all of the app containers in a Pod can start in parallel.
- Init containers can securely run utilities or custom code that would otherwise make an app container image less secure. By keeping unnecessary tools separate you can limit the attack surface of your app container image.

Examples

Here are some ideas for how to use init containers:

- Wait for a [Service](#) to be created, using a shell one-line command like:

```
for i in {1..100}; do sleep 1; if dig myservice; then exit 0; fi; done; exit 1
```

- Register this Pod with a remote server from the downward API with a command like:

```
curl -X POST http://$MANAGEMENT_SERVICE_HOST:$MANAGEMENT_SERVICE_PORT/register -d 'instance=$(<POD_NAME>)&ip=$(<POD_IP>)'
```

- Wait for some time before starting the app container with a command like

```
sleep 60
```

- Clone a Git repository into a [Volume](#)

- Place values into a configuration file and run a template tool to dynamically generate a configuration file for the main app container. For example, place the `POD_IP` value in a configuration and generate the main app configuration file using `Jinja`.

Init containers in use

This example defines a simple Pod that has two init containers. The first waits for `myservice`, and the second waits for `mydb`. Once both init containers complete, the Pod runs the app container from its `spec` section.

```
apiVersion: v1
kind: Pod
metadata:
  name: myapp-pod
  labels:
    app: myapp
spec:
  containers:
    - name: myapp-container
      image: busybox:1.28
      command: ['sh', '-c', 'echo The app is running! && sleep 3600']
  initContainers:
    - name: init-myservice
      image: busybox:1.28
      command: ['sh', '-c', "until nslookup myservice.$(cat /var/run/secrets/kubernetes.io/serviceaccount/namespace).svc.cluster.local; do echo waiting for myservice; sleep 2; done"]
    - name: init-mydb
      image: busybox:1.28
      command: ['sh', '-c', "until nslookup mydb.$(cat /var/run/secrets/kubernetes.io/serviceaccount/namespace).svc.cluster.local; do echo waiting for mydb; sleep 2; done"]
```

You can start this Pod by running:

```
kubectl apply -f myapp.yaml
```

```
pod/myapp-pod created
```

And check on its status with:

```
kubectl get -f myapp.yaml
```

NAME	READY	STATUS	RESTARTS	AGE
myapp-pod	0/1	Init:0/2	0	6m

or for more details:

```
kubectl describe -f myapp.yaml
```

Name:	myapp-pod
Namespace:	default
[...]	
Labels:	app=myapp

```

Status:          Pending
[...]
Init Containers:
  init-myservice:
[...]
    State:        Running
[...]
  init-mydb:
[...]
    State:        Waiting
    Reason:       PodInitializing
    Ready:        False
[...]
Containers:
  myapp-container:
[...]
    State:        Waiting
    Reason:       PodInitializing
    Ready:        False
[...]
Events:
  FirstSeen     LastSeen      Count   From
SubObjectPath           Type
Reason            Message
-----  -----
-----  -----
-----  -----
  16s           16s          1      {default-
scheduler }
Normal          Scheduled      Successfully assigned myapp-pod to
172.17.4.201
  16s           16s          1      {kubelet 172.17.4.201}
spec.initContainers{init-myservice}  Normal
Pulling          pulling image "busybox"
  13s           13s          1      {kubelet 172.17.4.201}
spec.initContainers{init-myservice}  Normal
Pulled           Successfully pulled image "busybox"
  13s           13s          1      {kubelet 172.17.4.201}
spec.initContainers{init-myservice}  Normal
Created          Created container with docker id 5ced34a04634;
Security:[seccomp=unconfined]
  13s           13s          1      {kubelet 172.17.4.201}
spec.initContainers{init-myservice}  Normal
Started          Started container with docker id 5ced34a04634

```

To see logs for the init containers in this Pod, run:

```

kubectl logs myapp-pod -c init-myservice # Inspect the first
init container
kubectl logs myapp-pod -c init-mydb          # Inspect the second
init container

```

At this point, those init containers will be waiting to discover Services named `mydb` and `myservice`.

Here's a configuration you can use to make those Services appear:

```
---  
apiVersion: v1  
kind: Service  
metadata:  
  name: myservice  
spec:  
  ports:  
    - protocol: TCP  
      port: 80  
      targetPort: 9376  
---  
apiVersion: v1  
kind: Service  
metadata:  
  name: mydb  
spec:  
  ports:  
    - protocol: TCP  
      port: 80  
      targetPort: 9377
```

To create the `mydb` and `myservice` services:

```
kubectl apply -f services.yaml
```

```
service/myservice created  
service/mydb created
```

You'll then see that those init containers complete, and that the `myapp-pod` Pod moves into the `Running` state:

```
kubectl get -f myapp.yaml
```

NAME	READY	STATUS	RESTARTS	AGE
myapp-pod	1/1	Running	0	9m

This simple example should provide some inspiration for you to create your own init containers. [What's next](#) contains a link to a more detailed example.

Detailed behavior

During Pod startup, the kubelet delays running init containers until the networking and storage are ready. Then the kubelet runs the Pod's init containers in the order they appear in the Pod's spec.

Each init container must exit successfully before the next container starts. If a container fails to start due to the runtime or exits with failure, it is retried

according to the Pod `restartPolicy`. However, if the Pod `restartPolicy` is set to Always, the init containers use `restartPolicy` OnFailure.

A Pod cannot be Ready until all init containers have succeeded. The ports on an init container are not aggregated under a Service. A Pod that is initializing is in the Pending state but should have a condition `Initialized` set to true.

If the Pod [restarts](#), or is restarted, all init containers must execute again.

Changes to the init container spec are limited to the container image field. Altering an init container image field is equivalent to restarting the Pod.

Because init containers can be restarted, retried, or re-executed, init container code should be idempotent. In particular, code that writes to files on `EmptyDirs` should be prepared for the possibility that an output file already exists.

Init containers have all of the fields of an app container. However, Kubernetes prohibits `readinessProbe` from being used because init containers cannot define readiness distinct from completion. This is enforced during validation.

Use `activeDeadlineSeconds` on the Pod and `livenessProbe` on the container to prevent init containers from failing forever. The active deadline includes init containers.

The name of each app and init container in a Pod must be unique; a validation error is thrown for any container sharing a name with another.

Resources

Given the ordering and execution for init containers, the following rules for resource usage apply:

- *The highest of any particular resource request or limit defined on all init containers is the effective init request/limit*
- *The Pod's effective request/limit for a resource is the higher of:*
 - *the sum of all app containers request/limit for a resource*
 - *the effective init request/limit for a resource*
- *Scheduling is done based on effective requests/limits, which means init containers can reserve resources for initialization that are not used during the life of the Pod.*
- *The QoS (quality of service) tier of the Pod's effective QoS tier is the QoS tier for init containers and app containers alike.*

Quota and limits are applied based on the effective Pod request and limit.

Pod level control groups (cgroups) are based on the effective Pod request and limit, the same as the scheduler.

Pod restart reasons

A Pod can restart, causing re-execution of init containers, for the following reasons:

- A user updates the Pod specification, causing the init container image to change. Any changes to the init container image restarts the Pod. App container image changes only restart the app container.
- The Pod infrastructure container is restarted. This is uncommon and would have to be done by someone with root access to nodes.
- All containers in a Pod are terminated while `restartPolicy` is set to `Always`, forcing a restart, and the init container completion record has been lost due to garbage collection.

What's next

- Read about [creating a Pod that has an init container](#)
- Learn how to [debug init containers](#)

Feedback

Was this page helpful?

Yes No

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

Last modified December 19, 2020 at 2:13 PM PST: [Fix some typo in init-containers.md \(21c575008\)](#)

[Edit this page](#) [Create child page](#) [Create an issue](#)

- [Understanding init containers](#)
 - [Differences from regular containers](#)
- [Using init containers](#)
 - [Examples](#)
- [Detailed behavior](#)
 - [Resources](#)
 - [Pod restart reasons](#)
- [What's next](#)

Pod Topology Spread Constraints

FEATURE STATE: Kubernetes v1.19 [stable]

You can use topology spread constraints to control how [Pods](#) are spread across your cluster among failure-domains such as regions, zones, nodes, and other user-defined topology domains. This can help to achieve high availability as well as efficient resource utilization.

Note: In versions of Kubernetes before v1.19, you must enable the `EvenPodsSpread` [feature gate](#) on the [API server](#) and the [scheduler](#) in order to use Pod topology spread constraints.

Prerequisites

Node Labels

Topology spread constraints rely on node labels to identify the topology domain(s) that each Node is in. For example, a Node might have labels: `node=node1,zone=us-east-1a,region=us-east-1`

Suppose you have a 4-node cluster with the following labels:

NAME	STATUS	ROLES	AGE	VERSION	LABELS
node1	Ready	<none>	4m26s	v1.16.0	<code>node=node1,zone=zoneA</code>
node2	Ready	<none>	3m58s	v1.16.0	<code>node=node2,zone=zoneA</code>
node3	Ready	<none>	3m17s	v1.16.0	<code>node=node3,zone=zoneB</code>
node4	Ready	<none>	2m43s	v1.16.0	<code>node=node4,zone=zoneB</code>

Then the cluster is logically viewed as below:

[JavaScript must be [enabled](#) to view content]

Instead of manually applying labels, you can also reuse the [well-known labels](#) that are created and populated automatically on most clusters.

Spread Constraints for Pods

API

The API field `pod.spec.topologySpreadConstraints` is defined as below:

```
apiVersion: v1
kind: Pod
metadata:
  name: mypod
spec:
  topologySpreadConstraints:
    - maxSkew: <integer>
      topologyKey: <string>
      whenUnsatisfiable: <string>
      labelSelector: <object>
```

You can define one or multiple `topologySpreadConstraint` to instruct the kube-scheduler how to place each incoming Pod in relation to the existing Pods across your cluster. The fields are:

- **maxSkew** describes the degree to which Pods may be unevenly distributed. It's the maximum permitted difference between the number of matching Pods in any two topology domains of a given topology type. It must be greater than zero. Its semantics differs according to the value of `whenUnsatisfiable`:
 - when `whenUnsatisfiable` equals to "DoNotSchedule", `maxSkew` is the maximum permitted difference between the number of matching pods in the target topology and the global minimum.
 - when `whenUnsatisfiable` equals to "ScheduleAnyway", scheduler gives higher precedence to topologies that would help reduce the skew.
- **topologyKey** is the key of node labels. If two Nodes are labelled with this key and have identical values for that label, the scheduler treats both Nodes as being in the same topology. The scheduler tries to place a balanced number of Pods into each topology domain.
- **whenUnsatisfiable** indicates how to deal with a Pod if it doesn't satisfy the spread constraint:
 - `DoNotSchedule` (default) tells the scheduler not to schedule it.
 - `ScheduleAnyway` tells the scheduler to still schedule it while prioritizing nodes that minimize the skew.
- **labelSelector** is used to find matching Pods. Pods that match this label selector are counted to determine the number of Pods in their corresponding topology domain. See [Label Selectors](#) for more details.

You can read more about this field by running `kubectl explain Pod.spec.topologySpreadConstraints`.

Example: One `TopologySpreadConstraint`

Suppose you have a 4-node cluster where 3 Pods labeled `foo:bar` are located in `node1`, `node2` and `node3` respectively:

[JavaScript must be [enabled](#) to view content]

If we want an incoming Pod to be evenly spread with existing Pods across zones, the spec can be given as:

[pods/topology-spread-constraints/one-constraint.yaml](#)
□

```
kind: Pod
apiVersion: v1
```

```

metadata:
  name: mypod
  labels:
    foo: bar
spec:
  topologySpreadConstraints:
  - maxSkew: 1
    topologyKey: zone
    whenUnsatisfiable: DoNotSchedule
    labelSelector:
      matchLabels:
        foo: bar
  containers:
  - name: pause
    image: k8s.gcr.io/pause:3.1

```

topologyKey: zone implies the even distribution will only be applied to the nodes which have label pair "zone:<any value>" present. **whenUnsatisfiable: DoNotSchedule** tells the scheduler to let it stay pending if the incoming Pod can't satisfy the constraint.

If the scheduler placed this incoming Pod into "zoneA", the Pods distribution would become [3, 1], hence the actual skew is 2 (3 - 1) - which violates **maxSkew: 1**. In this example, the incoming Pod can only be placed onto "zoneB":

[JavaScript must be [enabled](#) to view content]

OR

[JavaScript must be [enabled](#) to view content]

You can tweak the Pod spec to meet various kinds of requirements:

- Change **maxSkew** to a bigger value like "2" so that the incoming Pod can be placed onto "zoneA" as well.
- Change **topologyKey** to "node" so as to distribute the Pods evenly across nodes instead of zones. In the above example, if **maxSkew** remains "1", the incoming Pod can only be placed onto "node4".
- Change **whenUnsatisfiable: DoNotSchedule** to **whenUnsatisfiable: ScheduleAnyway** to ensure the incoming Pod to be always schedulable (suppose other scheduling APIs are satisfied). However, it's preferred to be placed onto the topology domain which has fewer matching Pods. (Be aware that this preferability is jointly normalized with other internal scheduling priorities like resource usage ratio, etc.)

Example: Multiple TopologySpreadConstraints

This builds upon the previous example. Suppose you have a 4-node cluster where 3 Pods labeled `foo:bar` are located in `node1`, `node2` and `node3` respectively:

[JavaScript must be [enabled](#) to view content]

You can use 2 `TopologySpreadConstraints` to control the Pods spreading on both zone and node:

[pods/topology-spread-constraints/two-constraints.yaml](#)

```
kind: Pod
apiVersion: v1
metadata:
  name: mypod
  labels:
    foo: bar
spec:
  topologySpreadConstraints:
  - maxSkew: 1
    topologyKey: zone
    whenUnsatisfiable: DoNotSchedule
    labelSelector:
      matchLabels:
        foo: bar
  - maxSkew: 1
    topologyKey: node
    whenUnsatisfiable: DoNotSchedule
    labelSelector:
      matchLabels:
        foo: bar
  containers:
  - name: pause
    image: k8s.gcr.io/pause:3.1
```

In this case, to match the first constraint, the incoming Pod can only be placed onto "zoneB"; while in terms of the second constraint, the incoming Pod can only be placed onto "node4". Then the results of 2 constraints are ANDed, so the only viable option is to place on "node4".

Multiple constraints can lead to conflicts. Suppose you have a 3-node cluster across 2 zones:

[JavaScript must be [enabled](#) to view content]

If you apply "two-constraints.yaml" to this cluster, you will notice "mypod" stays in Pending state. This is because: to satisfy the first constraint, "mypod" can only be put to "zoneB"; while in terms of the second constraint, "mypod" can only put to "node2". Then a joint result of "zoneB" and "node2" returns nothing.

To overcome this situation, you can either increase the `maxSkew` or modify one of the constraints to use `whenUnsatisfiable: ScheduleAnyway`.

Conventions

There are some implicit conventions worth noting here:

- Only the Pods holding the same namespace as the incoming Pod can be matching candidates.
- Nodes without `topologySpreadConstraints[*].topologyKey` present will be bypassed. It implies that:
 1. the Pods located on those nodes do not impact `maxSkew` calculation - in the above example, suppose "node1" does not have label "zone", then the 2 Pods will be disregarded, hence the incoming Pod will be scheduled into "zoneA".
 2. the incoming Pod has no chances to be scheduled onto this kind of nodes - in the above example, suppose a "node5" carrying label {z one-typo: zoneC} joins the cluster, it will be bypassed due to the absence of label key "zone".
- Be aware of what will happen if the incomingPod's `topologySpreadConstraints[*].labelSelector` doesn't match its own labels. In the above example, if we remove the incoming Pod's labels, it can still be placed onto "zoneB" since the constraints are still satisfied. However, after the placement, the degree of imbalance of the cluster remains unchanged - it's still zoneA having 2 Pods which hold label {foo:bar}, and zoneB having 1 Pod which holds label {foo:bar}. So if this is not what you expect, we recommend the workload's `topologySpreadConstraints[*].labelSelector` to match its own labels.
- If the incoming Pod has `spec.nodeSelector` or `spec.affinity.nodeAffinity` defined, nodes not matching them will be bypassed.

Suppose you have a 5-node cluster ranging from zoneA to zoneC:

[JavaScript must be [enabled](#) to view content]

[JavaScript must be [enabled](#) to view content]

and you know that "zoneC" must be excluded. In this case, you can compose the yaml as below, so that "mypod" will be placed onto "zoneB" instead of "zoneC". Similarly `spec.nodeSelector` is also respected.

[pods/topology-spread-constraints/one-constraint-with-nodeaffinity.yaml](#)



```
kind: Pod
apiVersion: v1
metadata:
  name: mypod
  labels:
    foo: bar
spec:
  topologySpreadConstraints:
  - maxSkew: 1
    topologyKey: zone
    whenUnsatisfiable: DoNotSchedule
    labelSelector:
      matchLabels:
        foo: bar
  affinity:
    nodeAffinity:
      requiredDuringSchedulingIgnoredDuringExecution:
        nodeSelectorTerms:
        - matchExpressions:
          - key: zone
            operator: NotIn
            values:
            - zoneC
  containers:
  - name: pause
    image: k8s.gcr.io/pause:3.1
```

Cluster-level default constraints

It is possible to set default topology spread constraints for a cluster. Default topology spread constraints are applied to a Pod if, and only if:

- It doesn't define any constraints in its `.spec.topologySpreadConstraints`.
- It belongs to a service, replication controller, replica set or stateful set.

Default constraints can be set as part of the PodTopologySpread plugin args in a [scheduling profile](#). The constraints are specified with the same [API above](#), except that labelSelector must be empty. The selectors are calculated from the services, replication controllers, replica sets or stateful sets that the Pod belongs to.

An example configuration might look like follows:

```
apiVersion: kubescheduler.config.k8s.io/v1beta1
kind: KubeSchedulerConfiguration

profiles:
  - pluginConfig:
      - name: PodTopologySpread
        args:
          defaultConstraints:
            - maxSkew: 1
              topologyKey: topology.kubernetes.io/zone
              whenUnsatisfiable: ScheduleAnyway
            defaultingType: List
```

Note: The score produced by default scheduling constraints might conflict with the score produced by the [SelectorSpread plugin](#). It is recommended that you disable this plugin in the scheduling profile when using default constraints for PodTopologySpread.

Internal default constraints

FEATURE STATE: Kubernetes v1.20 [beta]

With the DefaultPodTopologySpread feature gate, enabled by default, the legacy SelectorSpread plugin is disabled. kube-scheduler uses the following default topology constraints for the PodTopologySpread plugin configuration:

```
defaultConstraints:
  - maxSkew: 3
    topologyKey: "kubernetes.io/hostname"
    whenUnsatisfiable: ScheduleAnyway
  - maxSkew: 5
    topologyKey: "topology.kubernetes.io/zone"
    whenUnsatisfiable: ScheduleAnyway
```

Also, the legacy SelectorSpread plugin, which provides an equivalent behavior, is disabled.

Note:

If your nodes are not expected to have **both** `kubernetes.io/hostname` and `topology.kubernetes.io/zone` labels set, define your own constraints instead of using the Kubernetes defaults.

The `PodTopologySpread` plugin does not score the nodes that don't have the topology keys specified in the spreading constraints.

If you don't want to use the default Pod spreading constraints for your cluster, you can disable those defaults by setting `defaultingType` to `List` and leaving empty `defaultConstraints` in the `PodTopologySpread` plugin configuration:

```
apiVersion: kubescheduler.config.k8s.io/v1beta1
kind: KubeSchedulerConfiguration

profiles:
  - pluginConfig:
      name: PodTopologySpread
      args:
        defaultConstraints: []
        defaultingType: List
```

Comparison with PodAffinity/PodAntiAffinity

In Kubernetes, directives related to "Affinity" control how Pods are scheduled - more packed or more scattered.

- For `PodAffinity`, you can try to pack any number of Pods into qualifying topology domain(s)
- For `PodAntiAffinity`, only one Pod can be scheduled into a single topology domain.

For finer control, you can specify topology spread constraints to distribute Pods across different topology domains - to achieve either high availability or cost-saving. This can also help on rolling update workloads and scaling out replicas smoothly. See [Motivation](#) for more details.

Known Limitations

- Scaling down a Deployment may result in imbalanced Pods distribution.
- Pods matched on tainted nodes are respected. See [Issue 80921](#)

What's next

- [Blog: Introducing PodTopologySpread](#) explains `maxSkew` in details, as well as bringing up some advanced usage examples.

Feedback

Was this page helpful?

Yes No

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

Last modified November 02, 2020 at 11:03 AM PST: [Graduate default pod topology spread to beta \(ac3d7d564\)](#)

[Edit this page](#) [Create child page](#) [Create an issue](#)

- [Prerequisites](#)
 - [Node Labels](#)
- [Spread Constraints for Pods](#)
 - [API](#)
 - [Example: One TopologySpreadConstraint](#)
 - [Example: Multiple TopologySpreadConstraints](#)
 - [Conventions](#)
 - [Cluster-level default constraints](#)
- [Comparison with PodAffinity/PodAntiAffinity](#)
- [Known Limitations](#)
- [What's next](#)

Disruptions

This guide is for application owners who want to build highly available applications, and thus need to understand what types of disruptions can happen to Pods.

It is also for cluster administrators who want to perform automated cluster actions, like upgrading and autoscaling clusters.

Voluntary and involuntary disruptions

Pods do not disappear until someone (a person or a controller) destroys them, or there is an unavoidable hardware or system software error.

We call these unavoidable cases *involuntary disruptions* to an application. Examples are:

- a hardware failure of the physical machine backing the node
- cluster administrator deletes VM (instance) by mistake
- cloud provider or hypervisor failure makes VM disappear
- a kernel panic
- the node disappears from the cluster due to cluster network partition
- eviction of a pod due to the node being [out-of-resources](#).

Except for the *out-of-resources* condition, all these conditions should be familiar to most users; they are not specific to Kubernetes.

We call other cases *voluntary disruptions*. These include both actions initiated by the application owner and those initiated by a Cluster Administrator. Typical application owner actions include:

- deleting the deployment or other controller that manages the pod
- updating a deployment's pod template causing a restart
- directly deleting a pod (e.g. by accident)

Cluster administrator actions include:

- [Draining a node](#) for repair or upgrade.
- Draining a node from a cluster to scale the cluster down (learn about [Cluster Autoscaling](#)).
- Removing a pod from a node to permit something else to fit on that node.

These actions might be taken directly by the cluster administrator, or by automation run by the cluster administrator, or by your cluster hosting provider.

Ask your cluster administrator or consult your cloud provider or distribution documentation to determine if any sources of voluntary disruptions are enabled for your cluster. If none are enabled, you can skip creating Pod Disruption Budgets.

Caution: Not all voluntary disruptions are constrained by Pod Disruption Budgets. For example, deleting deployments or pods bypasses Pod Disruption Budgets.

Dealing with disruptions

Here are some ways to mitigate involuntary disruptions:

- Ensure your pod [requests the resources](#) it needs.
- Replicate your application if you need higher availability. (Learn about running replicated [stateless](#) and [stateful](#) applications.)
- For even higher availability when running replicated applications, spread applications across racks (using [anti-affinity](#)) or across zones (if using a [multi-zone cluster](#).)

The frequency of voluntary disruptions varies. On a basic Kubernetes cluster, there are no voluntary disruptions at all. However, your cluster administrator or hosting provider may run some additional services which cause voluntary disruptions. For example, rolling out node software updates can cause voluntary disruptions. Also, some implementations of cluster (node) autoscaling may cause voluntary disruptions to defragment and compact nodes. Your cluster administrator or hosting provider should have documented what level of voluntary disruptions, if any, to expect.

Pod disruption budgets

FEATURE STATE: Kubernetes v1.5 [beta]

Kubernetes offers features to help you run highly available applications even when you introduce frequent voluntary disruptions.

As an application owner, you can create a PodDisruptionBudget (PDB) for each application. A PDB limits the number of Pods of a replicated application that are down simultaneously from voluntary disruptions. For example, a quorum-based application would like to ensure that the number of replicas running is never brought below the number needed for a quorum. A web front end might want to ensure that the number of replicas serving load never falls below a certain percentage of the total.

Cluster managers and hosting providers should use tools which respect PodDisruptionBudgets by calling the [Eviction API](#) instead of directly deleting pods or deployments.

For example, the `kubectl drain` subcommand lets you mark a node as going out of service. When you run `kubectl drain`, the tool tries to evict all of the Pods on the Node you're taking out of service. The eviction request that `kubectl` submits on your behalf may be temporarily rejected, so the tool periodically retries all failed requests until all Pods on the target node are terminated, or until a configurable timeout is reached.

A PDB specifies the number of replicas that an application can tolerate having, relative to how many it is intended to have. For example, a Deployment which has a `.spec.replicas: 5` is supposed to have 5 pods at any given time. If its PDB allows for there to be 4 at a time, then the Eviction API will allow voluntary disruption of one (but not two) pods at a time.

The group of pods that comprise the application is specified using a label selector, the same as the one used by the application's controller (deployment, stateful-set, etc).

The "intended" number of pods is computed from the `.spec.replicas` of the workload resource that is managing those pods. The control plane discovers the owning workload resource by examining the `.metadata.ownerReferences` of the Pod.

PDBs cannot prevent [involuntary disruptions](#) from occurring, but they do count against the budget.

Pods which are deleted or unavailable due to a rolling upgrade to an application do count against the disruption budget, but workload resources (such as Deployment and StatefulSet) are not limited by PDBs when doing rolling upgrades. Instead, the handling of failures during application updates is configured in the spec for the specific workload resource.

When a pod is evicted using the eviction API, it is gracefully [terminated](#), honoring the `terminationGracePeriodSeconds` setting in its [PodSpec](#).)

PodDisruptionBudget example

Consider a cluster with 3 nodes, `node-1` through `node-3`. The cluster is running several applications. One of them has 3 replicas initially called `pod-a`, `pod-b`, and `pod-c`. Another, unrelated pod without a PDB, called `pod-x`, is also shown. Initially, the pods are laid out as follows:

node-1	node-2	node-3
pod-a available	pod-b available	pod-c available
pod-x available		

All 3 pods are part of a deployment, and they collectively have a PDB which requires there be at least 2 of the 3 pods to be available at all times.

For example, assume the cluster administrator wants to reboot into a new kernel version to fix a bug in the kernel. The cluster administrator first tries to drain `node-1` using the `kubectl drain` command. That tool tries to evict `pod-a` and `pod-x`. This succeeds immediately. Both pods go into the `terminating` state at the same time. This puts the cluster in this state:

node-1 draining	node-2	node-3
pod-a terminating	pod-b available	pod-c available
pod-x terminating		

The deployment notices that one of the pods is terminating, so it creates a replacement called `pod-d`. Since `node-1` is cordoned, it lands on another node. Something has also created `pod-y` as a replacement for `pod-x`.

(Note: for a StatefulSet, `pod-a`, which would be called something like `pod-0`, would need to terminate completely before its replacement, which is also called `pod-0` but has a different UID, could be created. Otherwise, the example applies to a StatefulSet as well.)

Now the cluster is in this state:

node-1 draining	node-2	node-3
pod-a terminating	pod-b available	pod-c available
pod-x terminating	pod-d starting	pod-y

At some point, the pods terminate, and the cluster looks like this:

node-1 drained	node-2	node-3
	pod-b available	pod-c available
	pod-d starting	pod-y

At this point, if an impatient cluster administrator tries to drain node-2 or node-3, the drain command will block, because there are only 2 available pods for the deployment, and its PDB requires at least 2. After some time passes, pod-d becomes available.

The cluster state now looks like this:

node-1 drained	node-2	node-3
	pod-b available	pod-c available
	pod-d available	pod-y

Now, the cluster administrator tries to drain node-2. The drain command will try to evict the two pods in some order, say pod-b first and then pod-d. It will succeed at evicting pod-b. But, when it tries to evict pod-d, it will be refused because that would leave only one pod available for the deployment.

The deployment creates a replacement for pod-b called pod-e. Because there are not enough resources in the cluster to schedule pod-e the drain will again block. The cluster may end up in this state:

node-1 drained	node-2	node-3	no node
	pod-b terminating	pod-c available	pod-e pending
	pod-d available	pod-y	

At this point, the cluster administrator needs to add a node back to the cluster to proceed with the upgrade.

You can see how Kubernetes varies the rate at which disruptions can happen, according to:

- how many replicas an application needs
- how long it takes to gracefully shutdown an instance
- how long it takes a new instance to start up
- the type of controller
- the cluster's resource capacity

Separating Cluster Owner and Application Owner Roles

Often, it is useful to think of the Cluster Manager and Application Owner as separate roles with limited knowledge of each other. This separation of responsibilities may make sense in these scenarios:

- *when there are many application teams sharing a Kubernetes cluster, and there is natural specialization of roles*
- *when third-party tools or services are used to automate cluster management*

Pod Disruption Budgets support this separation of roles by providing an interface between the roles.

If you do not have such a separation of responsibilities in your organization, you may not need to use Pod Disruption Budgets.

How to perform Disruptive Actions on your Cluster

If you are a Cluster Administrator, and you need to perform a disruptive action on all the nodes in your cluster, such as a node or system software upgrade, here are some options:

- *Accept downtime during the upgrade.*
- *Failover to another complete replica cluster.*
 - *No downtime, but may be costly both for the duplicated nodes and for human effort to orchestrate the switchover.*
- *Write disruption tolerant applications and use PDBs.*
 - *No downtime.*
 - *Minimal resource duplication.*
 - *Allows more automation of cluster administration.*
 - *Writing disruption-tolerant applications is tricky, but the work to tolerate voluntary disruptions largely overlaps with work to support autoscaling and tolerating involuntary disruptions.*

What's next

- *Follow steps to protect your application by [configuring a Pod Disruption Budget](#).*
- *Learn more about [draining nodes](#)*
- *Learn about [updating a deployment](#) including steps to maintain its availability during the rollout.*

Feedback

Was this page helpful?

Yes No

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

Last modified July 17, 2020 at 3:46 PM PST: [Replace reference to redirect entries \(1\) \(0bdcd44e6\)](#)

[Edit this page](#) [Create child page](#) [Create an issue](#)

- [Voluntary and involuntary disruptions](#)
- [Dealing with disruptions](#)
- [Pod disruption budgets](#)
- [PodDisruptionBudget example](#)
- [Separating Cluster Owner and Application Owner Roles](#)
- [How to perform Disruptive Actions on your Cluster](#)
- [What's next](#)

Ephemeral Containers

FEATURE STATE: Kubernetes v1.16 [alpha]

This page provides an overview of ephemeral containers: a special type of container that runs temporarily in an existing [Pod](#) to accomplish user-initiated actions such as troubleshooting. You use ephemeral containers to inspect services rather than to build applications.

Warning: Ephemeral containers are in early alpha state and are not suitable for production clusters. In accordance with the [Kubernetes Deprecation Policy](#), this alpha feature could change significantly in the future or be removed entirely.

Understanding ephemeral containers

[Pods](#) are the fundamental building block of Kubernetes applications. Since Pods are intended to be disposable and replaceable, you cannot add a container to a Pod once it has been created. Instead, you usually delete and replace Pods in a controlled fashion using [deployments](#).

Sometimes it's necessary to inspect the state of an existing Pod, however, for example to troubleshoot a hard-to-reproduce bug. In these cases you can run an ephemeral container in an existing Pod to inspect its state and run arbitrary commands.

What is an ephemeral container?

Ephemeral containers differ from other containers in that they lack guarantees for resources or execution, and they will never be automatically restarted, so they are not appropriate for building applications. Ephemeral containers are described using the same `ContainerSpec` as regular containers, but many fields are incompatible and disallowed for ephemeral containers.

- Ephemeral containers may not have ports, so fields such as `ports`, `livenessProbe`, `readinessProbe` are disallowed.
- Pod resource allocations are immutable, so setting `resources` is disallowed.
- For a complete list of allowed fields, see the [EphemeralContainer reference documentation](#).

Ephemeral containers are created using a special `ephemeralContainers` handler in the API rather than by adding them directly to `pod.spec`, so it's not possible to add an ephemeral container using `kubectl edit`.

Like regular containers, you may not change or remove an ephemeral container after you have added it to a Pod.

Uses for ephemeral containers

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.

In particular, [distroless images](#) enable you to deploy minimal container images that reduce attack surface and exposure to bugs and vulnerabilities. Since distroless images do not include a shell or any debugging utilities, it's difficult to troubleshoot distroless images using `kubectl exec` alone.

When using ephemeral containers, it's helpful to enable [process namespace sharing](#) so you can view processes in other containers.

See [Debugging with Ephemeral Debug Container](#) for examples of troubleshooting using ephemeral containers.

Ephemeral containers API

Note: The examples in this section require the `EphemeralContainers` feature gate to be enabled, and Kubernetes client and server version v1.16 or later.

The examples in this section demonstrate how ephemeral containers appear in the API. You would normally use `kubectl debug` or another `kubectl plugin` to automate these steps rather than invoking the API directly.

Ephemeral containers are created using the `ephemeralContainers` subresource of Pod, which can be demonstrated using `kubectl --raw`. First describe the ephemeral container to add as an `EphemeralContainers` list:

```
{  
    "apiVersion": "v1",  
    "kind": "EphemeralContainers",  
    "metadata": {  
        "name": "example-pod"  
    },  
    "ephemeralContainers": [ {  
        "command": [  
            "sh"  
        ],  
        "image": "busybox",  
        "imagePullPolicy": "IfNotPresent",  
        "name": "debugger",  
        "stdin": true,  
        "tty": true,  
        "terminationMessagePolicy": "File"  
    }]  
}
```

To update the ephemeral containers of the already running `example-pod`:

```
kubectl replace --raw /api/v1/namespaces/default/pods/example-pod/ephemeralcontainers -f ec.json
```

This will return the new list of ephemeral containers:

```
{  
    "kind": "EphemeralContainers",  
    "apiVersion": "v1",  
    "metadata": {  
        "name": "example-pod",  
        "namespace": "default",  
        "selfLink": "/api/v1/namespaces/default/pods/example-pod/ephemeralcontainers",  
        "uid": "a14a6d9b-62f2-4119-9d8e-e2ed6bc3a47c",  
        "resourceVersion": "15886",  
        "creationTimestamp": "2019-08-29T06:41:42Z"  
    },  
    "ephemeralContainers": [  
        {  
            "name": "debugger",  
            "image": "busybox",  
            "command": [  
                "sh"  
            ],  
            "resources": {  
            }  
        },  
        {  
            "name": "another-ephemeral-container",  
            "image": "nginx",  
            "command": [  
                "sh"  
            ]  
        }  
    ]  
}
```

```
        "terminationMessagePolicy": "File",
        "imagePullPolicy": "IfNotPresent",
        "stdin": true,
        "tty": true
    }
]
}
```

You can view the state of the newly created ephemeral container using `kubectl describe`:

```
kubectl describe pod example-pod
```

```
...
Ephemeral Containers:
  debugger:
    Container ID: docker://cf81908f149e7e9213d3c3644eda55c72efaff67652a2685c1146f0ce151e80f
      Image:          busybox
      Image ID:       docker-pullable://busybox@sha256:9f1003c480699be56815db0f8146ad2e22efa85129b5b5983
      d0e0fb52d9ab70
      Port:          <none>
      Host Port:     <none>
      Command:
        sh
      State:         Running
      Started:       Thu, 29 Aug 2019 06:42:21 +0000
      Ready:          False
      Restart Count: 0
      Environment:   <none>
      Mounts:        <none>
...

```

You can interact with the new ephemeral container in the same way as other containers using `kubectl attach`, `kubectl exec`, and `kubectl logs`, for example:

```
kubectl attach -it example-pod -c debugger
```

Feedback

Was this page helpful?

Yes No

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

Last modified November 09, 2020 at 1:46 PM PST: [Update kubectl debug docs for 1.20 release \(#24847\) \(179c821b0\)](#)

- [Understanding ephemeral containers](#)
 - [What is an ephemeral container?](#)
- [Uses for ephemeral containers](#)
- [Ephemeral containers API](#)

Workload Resources

[**Deployments**](#)

[**ReplicaSet**](#)

[**StatefulSets**](#)

[**DaemonSet**](#)

[**Jobs**](#)

[**Garbage Collection**](#)

[**TTL Controller for Finished Resources**](#)

[**CronJob**](#)

[**ReplicationController**](#)

Deployments

A Deployment provides declarative updates for [Pods](#) and [ReplicaSets](#).

You describe a desired state in a Deployment, and the Deployment Controller changes the actual state to the desired state at a controlled rate. You can define Deployments to create new ReplicaSets, or to remove existing Deployments and adopt all their resources with new Deployments.

Note: Do not manage ReplicaSets owned by a Deployment.

Consider opening an issue in the main Kubernetes repository if your use case is not covered below.

Use Case

The following are typical use cases for Deployments:

- [Create a Deployment to rollout a ReplicaSet](#). The ReplicaSet creates Pods in the background. Check the status of the rollout to see if it succeeds or not.

- [Declare the new state of the Pods](#) by updating the PodTemplateSpec of the Deployment. A new ReplicaSet is created and the Deployment manages moving the Pods from the old ReplicaSet to the new one at a controlled rate. Each new ReplicaSet updates the revision of the Deployment.
- [Rollback to an earlier Deployment revision](#) if the current state of the Deployment is not stable. Each rollback updates the revision of the Deployment.
- [Scale up the Deployment to facilitate more load](#).
- [Pause the Deployment](#) to apply multiple fixes to its PodTemplateSpec and then resume it to start a new rollout.
- [Use the status of the Deployment](#) as an indicator that a rollout has stuck.
- [Clean up older ReplicaSets](#) that you don't need anymore.

Creating a Deployment

The following is an example of a Deployment. It creates a ReplicaSet to bring up three nginx Pods:

[controllers/nginx-deployment.yaml](#)


```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-deployment
  labels:
    app: nginx
spec:
  replicas: 3
  selector:
    matchLabels:
      app: nginx
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
        - name: nginx
          image: nginx:1.14.2
          ports:
            - containerPort: 80
```

In this example:

- A Deployment named `nginx-deployment` is created, indicated by the `.metadata.name` field.

- The Deployment creates three replicated Pods, indicated by the `.spec.replicas` field.

- The `.spec.selector` field defines how the Deployment finds which Pods to manage. In this case, you simply select a label that is defined in the Pod template (`app: nginx`). However, more sophisticated selection rules are possible, as long as the Pod template itself satisfies the rule.

Note: The `.spec.selector.matchLabels` field is a map of `{key,value}` pairs. A single `{key,value}` in the `matchLabels` map is equivalent to an element of `matchExpressions`, whose `key` field is "key" the operator is "In", and the values array contains only "value". All of the requirements, from both `matchLabels` and `matchExpressions`, must be satisfied in order to match.

- The `template` field contains the following sub-fields:
 - The Pods are labeled `app: nginx` using the `.metadata.labels` field.
 - The Pod template's specification, or `.template.spec` field, indicates that the Pods run one container, `nginx`, which runs the `nginx` [Docker Hub](#) image at version 1.14.2.
 - Create one container and name it `nginx` using the `.spec.template.spec.containers[0].name` field.

Before you begin, make sure your Kubernetes cluster is up and running. Follow the steps given below to create the above Deployment:

1. Create the Deployment by running the following command:

```
kubectl apply -f https://k8s.io/examples/controllers/nginx-deployment.yaml
```

Note: You can specify the `--record` flag to write the command executed in the resource annotation `kubernetes.io/change-cause`. The recorded change is useful for future introspection. For example, to see the commands executed in each Deployment revision.

1. Run `kubectl get deployments` to check if the Deployment was created.

If the Deployment is still being created, the output is similar to the following:

NAME	READY	UP-TO-DATE	AVAILABLE	AGE
<code>nginx-deployment</code>	0/3	0	0	1s

When you inspect the Deployments in your cluster, the following fields are displayed:

- `NAME` lists the names of the Deployments in the namespace.

- *READY* displays how many replicas of the application are available to your users. It follows the pattern ready/desired.
- *UP-TO-DATE* displays the number of replicas that have been updated to achieve the desired state.
- *AVAILABLE* displays how many replicas of the application are available to your users.
- *AGE* displays the amount of time that the application has been running.

Notice how the number of desired replicas is 3 according to .spec.replicas field.

2. To see the Deployment rollout status, run `kubectl rollout status deployment/nginx-deployment`.

The output is similar to:

```
Waiting for rollout to finish: 2 out of 3 new replicas have
been updated...
deployment "nginx-deployment" successfully rolled out
```

3. Run the `kubectl get deployments` again a few seconds later. The output is similar to this:

NAME	READY	UP-TO-DATE	AVAILABLE	AGE
nginx-deployment	3/3	3	3	18s

Notice that the Deployment has created all three replicas, and all replicas are up-to-date (they contain the latest Pod template) and available.

4. To see the ReplicaSet (rs) created by the Deployment, run `kubectl get rs`. The output is similar to this:

NAME	DESIRED	CURRENT	READY	AGE
nginx-deployment-75675f5897	3	3	3	18s

ReplicaSet output shows the following fields:

- *NAME* lists the names of the ReplicaSets in the namespace.
- *DESIRED* displays the desired number of replicas of the application, which you define when you create the Deployment. This is the desired state.
- *CURRENT* displays how many replicas are currently running.
- *READY* displays how many replicas of the application are available to your users.
- *AGE* displays the amount of time that the application has been running.

Notice that the name of the ReplicaSet is always formatted as [DEPLOYMENT-NAME] - [RANDOM-STRING]. The random string is randomly generated and uses the pod-template-hash as a seed.

- To see the labels automatically generated for each Pod, run `kubectl get pods --show-labels`. The output is similar to:

NAME	READY	STATUS
RESTARTS	AGE	LABELS
nginx-deployment-75675f5897-7ci7o	1/1	Running
0	18s	app=nginx, pod-template-hash=3123191453
nginx-deployment-75675f5897-kzszej	1/1	Running
0	18s	app=nginx, pod-template-hash=3123191453
nginx-deployment-75675f5897-qqcnn	1/1	Running
0	18s	app=nginx, pod-template-hash=3123191453

The created ReplicaSet ensures that there are three nginx Pods.

Note:

You must specify an appropriate selector and Pod template labels in a Deployment (in this case, `app: nginx`).

Do not overlap labels or selectors with other controllers (including other Deployments and StatefulSets). Kubernetes doesn't stop you from overlapping, and if multiple controllers have overlapping selectors those controllers might conflict and behave unexpectedly.

Pod-template-hash label

Caution: Do not change this label.

The `pod-template-hash` label is added by the Deployment controller to every ReplicaSet that a Deployment creates or adopts.

This label ensures that child ReplicaSets of a Deployment do not overlap. It is generated by hashing the `PodTemplate` of the ReplicaSet and using the resulting hash as the label value that is added to the ReplicaSet selector, Pod template labels, and in any existing Pods that the ReplicaSet might have.

Updating a Deployment

Note: A Deployment's rollout is triggered if and only if the Deployment's Pod template (that is, `.spec.template`) is changed, for example if the labels or container images of the template are updated. Other updates, such as scaling the Deployment, do not trigger a rollout.

Follow the steps given below to update your Deployment:

- Let's update the nginx Pods to use the `nginx:1.16.1` image instead of the `nginx:1.14.2` image.

```
kubectl --record deployment.apps/nginx-deployment set image deployment.v1.apps/nginx-deployment nginx=nginx:1.16.1
```

or simply use the following command:

```
kubectl set image deployment/nginx-deployment nginx=nginx:1.16.1 --record
```

The output is similar to this:

```
deployment.apps/nginx-deployment image updated
```

Alternatively, you can edit the Deployment and change `.spec.template.spec.containers[0].image` from `nginx:1.14.2` to `nginx:1.16.1`:

```
kubectl edit deployment.v1.apps/nginx-deployment
```

The output is similar to this:

```
deployment.apps/nginx-deployment edited
```

2. To see the rollout status, run:

```
kubectl rollout status deployment/nginx-deployment
```

The output is similar to this:

```
Waiting for rollout to finish: 2 out of 3 new replicas have been updated...
```

or

```
deployment "nginx-deployment" successfully rolled out
```

Get more details on your updated Deployment:

- After the rollout succeeds, you can view the Deployment by running `kubectl get deployments`. The output is similar to this:

NAME	READY	UP-TO-DATE	AVAILABLE	AGE
nginx-deployment	3/3	3	3	36s

- Run `kubectl get rs` to see that the Deployment updated the Pods by creating a new ReplicaSet and scaling it up to 3 replicas, as well as scaling down the old ReplicaSet to 0 replicas.

```
kubectl get rs
```

The output is similar to this:

NAME	DESIRED	CURRENT	READY	AGE
nginx-deployment-1564180365	3	3	3	6s
nginx-deployment-2035384211	0	0	0	36s

- Running `get pods` should now show only the new Pods:

```
kubectl get pods
```

The output is similar to this:

NAME	READY	STATUS
RESTARTS	AGE	
nginx-deployment-1564180365-khku8	1/1	Running
0 14s		
nginx-deployment-1564180365-nacti	1/1	Running
0 14s		
nginx-deployment-1564180365-z9gth	1/1	Running
0 14s		

Next time you want to update these Pods, you only need to update the Deployment's Pod template again.

Deployment ensures that only a certain number of Pods are down while they are being updated. By default, it ensures that at least 75% of the desired number of Pods are up (25% max unavailable).

Deployment also ensures that only a certain number of Pods are created above the desired number of Pods. By default, it ensures that at most 125% of the desired number of Pods are up (25% max surge).

For example, if you look at the above Deployment closely, you will see that it first created a new Pod, then deleted some old Pods, and created new ones. It does not kill old Pods until a sufficient number of new Pods have come up, and does not create new Pods until a sufficient number of old Pods have been killed. It makes sure that at least 2 Pods are available and that at max 4 Pods in total are available.

- Get details of your Deployment:

```
kubectl describe deployments
```

The output is similar to this:

```
Name: nginx-deployment
Namespace: default
CreationTimestamp: Thu, 30 Nov 2017 10:56:25 +0000
Labels: app=nginx
Annotations: deployment.kubernetes.io/revision=2
Selector: app=nginx
Replicas: 3 desired | 3 updated | 3 total | 3
available | 0 unavailable
StrategyType: RollingUpdate
MinReadySeconds: 0
RollingUpdateStrategy: 25% max unavailable, 25% max surge
Pod Template:
  Labels: app=nginx
  Containers:
    nginx:
      Image: nginx:1.16.1
```

```

  Port:          80/TCP
  Environment:   <none>
  Mounts:        <none>
  Volumes:       <none>
Conditions:
  Type           Status  Reason
  ----
  Available     True    MinimumReplicasAvailable
  Progressing   True    NewReplicaSetAvailable
OldReplicaSets: <none>
NewReplicaSet:  nginx-deployment-1564180365 (3/3 replicas
created)
Events:
  Type      Reason          Age      From
Message
  ----      -----          ----      ----
  Normal   ScalingReplicaSet 2m      deployment-controller
Scaled up replica set nginx-deployment-2035384211 to 3
  Normal   ScalingReplicaSet 24s     deployment-controller
Scaled up replica set nginx-deployment-1564180365 to 1
  Normal   ScalingReplicaSet 22s     deployment-controller
Scaled down replica set nginx-deployment-2035384211 to 2
  Normal   ScalingReplicaSet 22s     deployment-controller
Scaled up replica set nginx-deployment-1564180365 to 2
  Normal   ScalingReplicaSet 19s     deployment-controller
Scaled down replica set nginx-deployment-2035384211 to 1
  Normal   ScalingReplicaSet 19s     deployment-controller
Scaled up replica set nginx-deployment-1564180365 to 3
  Normal   ScalingReplicaSet 14s     deployment-controller
Scaled down replica set nginx-deployment-2035384211 to 0

```

Here you see that when you first created the Deployment, it created a ReplicaSet (nginx-deployment-2035384211) and scaled it up to 3 replicas directly. When you updated the Deployment, it created a new ReplicaSet (nginx-deployment-1564180365) and scaled it up to 1 and then scaled down the old ReplicaSet to 2, so that at least 2 Pods were available and at most 4 Pods were created at all times. It then continued scaling up and down the new and the old ReplicaSet, with the same rolling update strategy. Finally, you'll have 3 available replicas in the new ReplicaSet, and the old ReplicaSet is scaled down to 0.

Rollover (aka multiple updates in-flight)

Each time a new Deployment is observed by the Deployment controller, a ReplicaSet is created to bring up the desired Pods. If the Deployment is updated, the existing ReplicaSet that controls Pods whose labels match `.spec.selector` but whose template does not match `.spec.template` are scaled down. Eventually, the new ReplicaSet is scaled to `.spec.replicas` and all old ReplicaSets are scaled to 0.

If you update a Deployment while an existing rollout is in progress, the Deployment creates a new ReplicaSet as per the update and start scaling that up, and rolls over the ReplicaSet that it was scaling up previously -- it will add it to its list of old ReplicaSets and start scaling it down.

For example, suppose you create a Deployment to create 5 replicas of nginx:1.14.2, but then update the Deployment to create 5 replicas of nginx:1.16.1, when only 3 replicas of nginx:1.14.2 had been created. In that case, the Deployment immediately starts killing the 3 nginx:1.14.2 Pods that it had created, and starts creating nginx:1.16.1 Pods. It does not wait for the 5 replicas of nginx:1.14.2 to be created before changing course.

Label selector updates

It is generally discouraged to make label selector updates and it is suggested to plan your selectors up front. In any case, if you need to perform a label selector update, exercise great caution and make sure you have grasped all of the implications.

Note: *In API version apps/v1, a Deployment's label selector is immutable after it gets created.*

- Selector additions require the Pod template labels in the Deployment spec to be updated with the new label too, otherwise a validation error is returned. This change is a non-overlapping one, meaning that the new selector does not select ReplicaSets and Pods created with the old selector, resulting in orphaning all old ReplicaSets and creating a new ReplicaSet.
- Selector updates changes the existing value in a selector key -- result in the same behavior as additions.
- Selector removals removes an existing key from the Deployment selector -- do not require any changes in the Pod template labels. Existing ReplicaSets are not orphaned, and a new ReplicaSet is not created, but note that the removed label still exists in any existing Pods and ReplicaSets.

Rolling Back a Deployment

Sometimes, you may want to rollback a Deployment; for example, when the Deployment is not stable, such as crash looping. By default, all of the Deployment's rollout history is kept in the system so that you can rollback anytime you want (you can change that by modifying revision history limit).

Note: *A Deployment's revision is created when a Deployment's rollout is triggered. This means that the new revision is created if and only if the Deployment's Pod template (.spec.template) is changed, for example if you update the labels or container images of the template. Other updates, such as scaling the Deployment, do not create a Deployment revision, so that you can facilitate simultaneous manual- or auto-scaling. This means that when you*

roll back to an earlier revision, only the Deployment's Pod template part is rolled back.

- Suppose that you made a typo while updating the Deployment, by putting the image name as `nginx:1.161` instead of `nginx:1.16.1`:

```
kubectl set image deployment.v1.apps/nginx-deployment nginx=n  
ginx:1.161 --record=true
```

The output is similar to this:

```
deployment.apps/nginx-deployment image updated
```

- The rollout gets stuck. You can verify it by checking the rollout status:

```
kubectl rollout status deployment/nginx-deployment
```

The output is similar to this:

```
Waiting for rollout to finish: 1 out of 3 new replicas have  
been updated...
```

- Press `Ctrl-C` to stop the above rollout status watch. For more information on stuck rollouts, [read more here](#).
- You see that the number of old replicas (`nginx-deployment-1564180365` and `nginx-deployment-2035384211`) is 2, and new replicas (`nginx-deployment-3066724191`) is 1.

```
kubectl get rs
```

The output is similar to this:

NAME	DESIRED	CURRENT	READY	AGE
<code>nginx-deployment-1564180365</code>	3	3	3	25s
<code>nginx-deployment-2035384211</code>	0	0	0	36s
<code>nginx-deployment-3066724191</code>	1	1	0	6s

- Looking at the Pods created, you see that 1 Pod created by new ReplicaSet is stuck in an image pull loop.

```
kubectl get pods
```

The output is similar to this:

READY			
STATUS	RESTARTS	AGE	
<code>nginx-deployment-1564180365-70iae</code>	0	25s	1/1
<code>Running</code>	0	25s	
<code>nginx-deployment-1564180365-jbqqa</code>	0	25s	1/1
<code>Running</code>	0	25s	
<code>nginx-deployment-1564180365-hysrc</code>	0	25s	1/1
<code>Running</code>	0	25s	

```
nginx-deployment-3066724191-08mng    0/1
ImagePullBackOff          0             6s
```

Note: The Deployment controller stops the bad rollout automatically, and stops scaling up the new ReplicaSet. This depends on the rollingUpdate parameters (`maxUnavailable` specifically) that you have specified. Kubernetes by default sets the value to 25%.

- Get the description of the Deployment:

```
kubectl describe deployment
```

The output is similar to this:

```
Name:           nginx-deployment
Namespace:      default
CreationTimestamp: Tue, 15 Mar 2016 14:48:04 -0700
Labels:         app=nginx
Selector:       app=nginx
Replicas:       3 desired | 1 updated | 4 total | 3
available | 1 unavailable
StrategyType:   RollingUpdate
MinReadySeconds: 0
RollingUpdateStrategy: 25% max unavailable, 25% max surge
Pod Template:
  Labels: app=nginx
  Containers:
    nginx:
      Image:      nginx:1.161
      Port:       80/TCP
      Host Port:  0/TCP
      Environment: <none>
      Mounts:     <none>
      Volumes:    <none>
  Conditions:
    Type        Status  Reason
    ----        ----  -----
    Available   True    MinimumReplicasAvailable
    Progressing True    ReplicaSetUpdated
OldReplicaSets:  nginx-deployment-1564180365 (3/3
replicas created)
NewReplicaSet:   nginx-deployment-3066724191 (1/1
replicas created)
Events:
  FirstSeen  LastSeen  Count  From            SubObjectPath  Type            Reason            Message
  ----       ----       ---  ----            ----          ----            ----            ----
  1m         1m         1     {deployment-
controller }          Normal   ScalingReplicaSet
Scaled up replica set nginx-deployment-2035384211 to 3
```

```

  22s      22s      1      {deployment-
controller }           Normal   ScalingReplicaSet
Scaled up replica set nginx-deployment-1564180365 to 1
  22s      22s      1      {deployment-
controller }           Normal   ScalingReplicaSet
Scaled down replica set nginx-deployment-2035384211 to 2
  22s      22s      1      {deployment-
controller }           Normal   ScalingReplicaSet
Scaled up replica set nginx-deployment-1564180365 to 2
  21s      21s      1      {deployment-
controller }           Normal   ScalingReplicaSet
Scaled down replica set nginx-deployment-2035384211 to 1
  21s      21s      1      {deployment-
controller }           Normal   ScalingReplicaSet
Scaled up replica set nginx-deployment-1564180365 to 3
  13s      13s      1      {deployment-
controller }           Normal   ScalingReplicaSet
Scaled down replica set nginx-deployment-2035384211 to 0
  13s      13s      1      {deployment-
controller }           Normal   ScalingReplicaSet
Scaled up replica set nginx-deployment-3066724191 to 1

```

To fix this, you need to rollback to a previous revision of Deployment that is stable.

Checking Rollout History of a Deployment

Follow the steps given below to check the rollout history:

1. First, check the revisions of this Deployment:

```
kubectl rollout history deployment.v1.apps/nginx-deployment
```

The output is similar to this:

```

deployments "nginx-deployment"
REVISION  CHANGE-CAUSE
1          kubectl apply --filename=https://k8s.io/examples/
controllers/nginx-deployment.yaml --record=true
2          kubectl set image deployment.v1.apps/nginx-
deployment nginx=nginx:1.16.1 --record=true
3          kubectl set image deployment.v1.apps/nginx-
deployment nginx=nginx:1.161 --record=true

```

CHANGE-CAUSE is copied from the Deployment annotation `kubernetes.io/change-cause` to its revisions upon creation. You can specify the CHANGE-CAUSE message by:

- Annotating the Deployment with `kubectl annotate deployment.v1.apps/nginx-deployment kubernetes.io/change-cause="image updated to 1.16.1"`
- Append the `--record` flag to save the `kubectl` command that is making changes to the resource.

- Manually editing the manifest of the resource.
2. To see the details of each revision, run:

```
kubectl rollout history deployment.v1.apps/nginx-deployment  
--revision=2
```

The output is similar to this:

```
deployments "nginx-deployment" revision 2  
Labels: app=nginx  
          pod-template-hash=1159050644  
Annotations: kubernetes.io/change-cause=kubectl set image  
deployment.v1.apps/nginx-deployment nginx=nginx:1.16.1 --  
record=true  
Containers:  
  nginx:  
    Image:      nginx:1.16.1  
    Port:       80/TCP  
    QoS Tier:  
      cpu:       BestEffort  
      memory:   BestEffort  
    Environment Variables: <none>  
  No volumes.
```

Rolling Back to a Previous Revision

Follow the steps given below to rollback the Deployment from the current version to the previous version, which is version 2.

1. Now you've decided to undo the current rollout and rollback to the previous revision:

```
kubectl rollout undo deployment.v1.apps/nginx-deployment
```

The output is similar to this:

```
deployment.apps/nginx-deployment rolled back
```

Alternatively, you can rollback to a specific revision by specifying it with `--to-revision`:

```
kubectl rollout undo deployment.v1.apps/nginx-deployment --  
to-revision=2
```

The output is similar to this:

```
deployment.apps/nginx-deployment rolled back
```

For more details about rollout related commands, read [kubectl rollout](#).

The Deployment is now rolled back to a previous stable revision. As you can see, a DeploymentRollback event for rolling back to revision 2 is generated from Deployment controller.

2. Check if the rollback was successful and the Deployment is running as expected, run:

```
kubectl get deployment nginx-deployment
```

The output is similar to this:

NAME	READY	UP-TO-DATE	AVAILABLE	AGE
nginx-deployment	3/3	3	3	30m

3. Get the description of the Deployment:

```
kubectl describe deployment nginx-deployment
```

The output is similar to this:

```
Name: nginx-deployment
Namespace: default
CreationTimestamp: Sun, 02 Sep 2018 18:17:55 -0500
Labels: app=nginx
Annotations: deployment.kubernetes.io/revision=4
kubernetes.io/change-cause=kubectl
set image deployment.v1.apps/nginx-deployment nginx=nginx:
1.16.1 --record=true
Selector: app=nginx
Replicas: 3 desired | 3 updated | 3 total | 3
available | 0 unavailable
StrategyType: RollingUpdate
MinReadySeconds: 0
RollingUpdateStrategy: 25% max unavailable, 25% max surge
Pod Template:
  Labels:  app=nginx
  Containers:
    nginx:
      Image:      nginx:1.16.1
      Port:       80/TCP
      Host Port:  0/TCP
      Environment: <none>
      Mounts:     <none>
      Volumes:    <none>
  Conditions:
    Type        Status  Reason
    ----        ----  -----
    Available   True   MinimumReplicasAvailable
    Progressing True   NewReplicaSetAvailable
  OldReplicaSets: <none>
  NewReplicaSet:  nginx-deployment-c4747d96c (3/3 replicas
created)
  Events:     
```

Type	Reason	Age	From
Message			
---	-----	-----	-----
Normal	ScalingReplicaSet	12m	deployment-controller
Scaled up	replica set nginx-deployment-75675f5897 to 3		
Normal	ScalingReplicaSet	11m	deployment-controller
Scaled up	replica set nginx-deployment-c4747d96c to 1		
Normal	ScalingReplicaSet	11m	deployment-controller
Scaled down	replica set nginx-deployment-75675f5897 to 2		
Normal	ScalingReplicaSet	11m	deployment-controller
Scaled up	replica set nginx-deployment-c4747d96c to 2		
Normal	ScalingReplicaSet	11m	deployment-controller
Scaled down	replica set nginx-deployment-75675f5897 to 1		
Normal	ScalingReplicaSet	11m	deployment-controller
Scaled up	replica set nginx-deployment-c4747d96c to 3		
Normal	ScalingReplicaSet	11m	deployment-controller
Scaled down	replica set nginx-deployment-75675f5897 to 0		
Normal	ScalingReplicaSet	11m	deployment-controller
Scaled up	replica set nginx-deployment-595696685f to 1		
Normal	DeploymentRollback	15s	deployment-controller
Rolled back deployment "nginx-deployment" to revision 2			
Normal	ScalingReplicaSet	15s	deployment-controller
Scaled down	replica set nginx-deployment-595696685f to 0		

Scaling a Deployment

You can scale a Deployment by using the following command:

```
kubectl scale deployment.v1.apps/nginx-deployment --replicas=10
```

The output is similar to this:

```
deployment.apps/nginx-deployment scaled
```

Assuming [horizontal Pod autoscaling](#) is enabled in your cluster, you can setup an autoscaler for your Deployment and choose the minimum and maximum number of Pods you want to run based on the CPU utilization of your existing Pods.

```
kubectl autoscale deployment.v1.apps/nginx-deployment --min=10 --max=15 --cpu-percent=80
```

The output is similar to this:

```
deployment.apps/nginx-deployment scaled
```

Proportional scaling

RollingUpdate Deployments support running multiple versions of an application at the same time. When you or an autoscaler scales a RollingUpdate Deployment that is in the middle of a rollout (either in

progress or paused), the Deployment controller balances the additional replicas in the existing active ReplicaSets (ReplicaSets with Pods) in order to mitigate risk. This is called proportional scaling.

For example, you are running a Deployment with 10 replicas, `maxSurge=3`, and `maxUnavailable=2`.

- Ensure that the 10 replicas in your Deployment are running.

```
kubectl get deploy
```

The output is similar to this:

NAME	DESIRED	CURRENT	UP-TO-DATE
AVAILABLE	AGE		
nginx-deployment	10	10	10
10	50s		

- You update to a new image which happens to be unresolvable from inside the cluster.

```
kubectl set image deployment.v1.apps/nginx-deployment nginx=n  
ginx:sometag
```

The output is similar to this:

```
deployment.apps/nginx-deployment image updated
```

- The image update starts a new rollout with ReplicaSet `nginx-deployment-1989198191`, but it's blocked due to the `maxUnavailable` requirement that you mentioned above. Check out the rollout status:

```
kubectl get rs
```

The output is similar to this:

NAME	DESIRED	CURRENT	READY
AGE			
nginx-deployment-1989198191	5	5	0
9s			
nginx-deployment-618515232	8	8	8
1m			

- Then a new scaling request for the Deployment comes along. The autoscaler increments the Deployment replicas to 15. The Deployment controller needs to decide where to add these new 5 replicas. If you weren't using proportional scaling, all 5 of them would be added in the new ReplicaSet. With proportional scaling, you spread the additional replicas across all ReplicaSets. Bigger proportions go to the ReplicaSets with the most replicas and lower proportions go to ReplicaSets with less replicas. Any leftovers are added to the ReplicaSet with the most replicas. ReplicaSets with zero replicas are not scaled up.

In our example above, 3 replicas are added to the old ReplicaSet and 2 replicas are added to the new ReplicaSet. The rollout process should eventually move all replicas to the new ReplicaSet, assuming the new replicas become healthy. To confirm this, run:

```
kubectl get deploy
```

The output is similar to this:

NAME	DESIRED	CURRENT	UP-TO-DATE
AVAILABLE	AGE		
nginx-deployment	15	18	7
8	7m		

The rollout status confirms how the replicas were added to each ReplicaSet.

```
kubectl get rs
```

The output is similar to this:

NAME	DESIRED	CURRENT	READY	AGE
nginx-deployment-1989198191	7	7	0	7m
nginx-deployment-618515232	11	11	11	7m

Pausing and Resuming a Deployment

You can pause a Deployment before triggering one or more updates and then resume it. This allows you to apply multiple fixes in between pausing and resuming without triggering unnecessary rollouts.

- For example, with a Deployment that was just created: Get the Deployment details:

```
kubectl get deploy
```

The output is similar to this:

NAME	DESIRED	CURRENT	UP-TO-DATE	AVAILABLE	AGE
nginx	3	3	3	3	1m

Get the rollout status:

```
kubectl get rs
```

The output is similar to this:

NAME	DESIRED	CURRENT	READY	AGE
nginx-2142116321	3	3	3	1m

- Pause by running the following command:

```
kubectl rollout pause deployment.v1.apps/nginx-deployment
```

The output is similar to this:

```
deployment.apps/nginx-deployment paused
```

- Then update the image of the Deployment:

```
kubectl set image deployment.v1.apps/nginx-deployment nginx=n  
ginx:1.16.1
```

The output is similar to this:

```
deployment.apps/nginx-deployment image updated
```

- Notice that no new rollout started:

```
kubectl rollout history deployment.v1.apps/nginx-deployment
```

The output is similar to this:

```
deployments "nginx"  
REVISION CHANGE-CAUSE  
1 <none>
```

- Get the rollout status to ensure that the Deployment is updated successfully:

```
kubectl get rs
```

The output is similar to this:

NAME	DESIRED	CURRENT	READY	AGE
nginx-2142116321	3	3	3	2m

- You can make as many updates as you wish, for example, update the resources that will be used:

```
kubectl set resources deployment.v1.apps/nginx-deployment -  
c=nginx --limits=cpu=200m,mem=512Mi
```

The output is similar to this:

```
deployment.apps/nginx-deployment resource requirements  
updated
```

The initial state of the Deployment prior to pausing it will continue its function, but new updates to the Deployment will not have any effect as long as the Deployment is paused.

- Eventually, resume the Deployment and observe a new ReplicaSet coming up with all the new updates:

```
kubectl rollout resume deployment.v1.apps/nginx-deployment
```

The output is similar to this:

```
deployment.apps/nginx-deployment resumed
```

- Watch the status of the rollout until it's done.

```
kubectl get rs -w
```

The output is similar to this:

NAME	DESIRED	CURRENT	READY	AGE
nginx-2142116321	2	2	2	2m
nginx-3926361531	2	2	0	6s
nginx-3926361531	2	2	1	18s
nginx-2142116321	1	2	2	2m
nginx-2142116321	1	2	2	2m
nginx-3926361531	3	2	1	18s
nginx-3926361531	3	2	1	18s
nginx-2142116321	1	1	1	2m
nginx-3926361531	3	3	1	18s
nginx-3926361531	3	3	2	19s
nginx-2142116321	0	1	1	2m
nginx-2142116321	0	1	1	2m
nginx-2142116321	0	0	0	2m
nginx-3926361531	3	3	3	20s

- Get the status of the latest rollout:

```
kubectl get rs
```

The output is similar to this:

NAME	DESIRED	CURRENT	READY	AGE
nginx-2142116321	0	0	0	2m
nginx-3926361531	3	3	3	28s

Note: You cannot rollback a paused Deployment until you resume it.

Deployment status

A Deployment enters various states during its lifecycle. It can be [progressing](#) while rolling out a new ReplicaSet, it can be [complete](#), or it can [fail to progress](#).

Progressing Deployment

Kubernetes marks a Deployment as progressing when one of the following tasks is performed:

- The Deployment creates a new ReplicaSet.
- The Deployment is scaling up its newest ReplicaSet.
- The Deployment is scaling down its older ReplicaSet(s).
- New Pods become ready or available (ready for at least [MinReadySeconds](#)).

You can monitor the progress for a Deployment by using `kubectl rollout status`.

Complete Deployment

Kubernetes marks a Deployment as complete when it has the following characteristics:

- All of the replicas associated with the Deployment have been updated to the latest version you've specified, meaning any updates you've requested have been completed.
- All of the replicas associated with the Deployment are available.
- No old replicas for the Deployment are running.

You can check if a Deployment has completed by using `kubectl rollout status`. If the rollout completed successfully, `kubectl rollout status` returns a zero exit code.

```
kubectl rollout status deployment/nginx-deployment
```

The output is similar to this:

```
Waiting for rollout to finish: 2 of 3 updated replicas are
available...
deployment "nginx-deployment" successfully rolled out
```

and the exit status from `kubectl rollout` is 0 (success):

```
echo $?
```

```
0
```

Failed Deployment

Your Deployment may get stuck trying to deploy its newest ReplicaSet without ever completing. This can occur due to some of the following factors:

- Insufficient quota
- Readiness probe failures
- Image pull errors
- Insufficient permissions
- Limit ranges
- Application runtime misconfiguration

One way you can detect this condition is to specify a deadline parameter in your Deployment spec: ([.spec.progressDeadlineSeconds](#)). `.spec.progressDeadlineSeconds` denotes the number of seconds the Deployment controller waits before indicating (in the Deployment status) that the Deployment progress has stalled.

The following `kubectl` command sets the `spec` with `progressDeadlineSeconds` to make the controller report lack of progress for a Deployment after 10 minutes:

```
kubectl patch deployment.v1.apps/nginx-deployment -p '{"spec": {"progressDeadlineSeconds":600}}'
```

The output is similar to this:

```
deployment.apps/nginx-deployment patched
```

Once the deadline has been exceeded, the Deployment controller adds a `DeploymentCondition` with the following attributes to the Deployment's `.status.conditions`:

- `Type=Progressing`
- `Status=False`
- `Reason=ProgressDeadlineExceeded`

See the [Kubernetes API conventions](#) for more information on status conditions.

Note: Kubernetes takes no action on a stalled Deployment other than to report a status condition with `Reason=ProgressDeadlineExceeded`. Higher level orchestrators can take advantage of it and act accordingly, for example, rollback the Deployment to its previous version.

Note: If you pause a Deployment, Kubernetes does not check progress against your specified deadline. You can safely pause a Deployment in the middle of a rollout and resume without triggering the condition for exceeding the deadline.

You may experience transient errors with your Deployments, either due to a low timeout that you have set or due to any other kind of error that can be treated as transient. For example, let's suppose you have insufficient quota. If you describe the Deployment you will notice the following section:

```
kubectl describe deployment nginx-deployment
```

The output is similar to this:

```
<...>
Conditions:
  Type        Status  Reason
  ----        ----   -----
  Available   True    MinimumReplicasAvailable
  Progressing True    ReplicaSetUpdated
  ReplicaFailure  True    FailedCreate
<...>
```

If you run `kubectl get deployment nginx-deployment -o yaml`, the Deployment status is similar to this:

```

status:
  availableReplicas: 2
  conditions:
    - lastTransitionTime: 2016-10-04T12:25:39Z
      lastUpdateTime: 2016-10-04T12:25:39Z
      message: Replica set "nginx-deployment-4262182780" is
      progressing.
      reason: ReplicaSetUpdated
      status: "True"
      type: Progressing
    - lastTransitionTime: 2016-10-04T12:25:42Z
      lastUpdateTime: 2016-10-04T12:25:42Z
      message: Deployment has minimum availability.
      reason: MinimumReplicasAvailable
      status: "True"
      type: Available
    - lastTransitionTime: 2016-10-04T12:25:39Z
      lastUpdateTime: 2016-10-04T12:25:39Z
      message: 'Error creating: pods "nginx-
      deployment-4262182780-' is forbidden: exceeded quota:
      object-counts, requested: pods=1, used: pods=3, limited:
      pods=2'
      reason: FailedCreate
      status: "True"
      type: ReplicaFailure
      observedGeneration: 3
      replicas: 2
      unavailableReplicas: 2

```

Eventually, once the Deployment progress deadline is exceeded, Kubernetes updates the status and the reason for the Progressing condition:

Conditions:

Type	Status	Reason
---	---	---
Available	True	MinimumReplicasAvailable
Progressing	False	ProgressDeadlineExceeded
ReplicaFailure	True	FailedCreate

You can address an issue of insufficient quota by scaling down your Deployment, by scaling down other controllers you may be running, or by increasing quota in your namespace. If you satisfy the quota conditions and the Deployment controller then completes the Deployment rollout, you'll see the Deployment's status update with a successful condition (Status=True and Reason>NewReplicaSetAvailable).

Conditions:

Type	Status	Reason
---	---	---
Available	True	MinimumReplicasAvailable
Progressing	True	NewReplicaSetAvailable

`Type=Available` with `Status=True` means that your Deployment has minimum availability. Minimum availability is dictated by the parameters specified in the deployment strategy. `Type=Progressing` with `Status=True` means that your Deployment is either in the middle of a rollout and it is progressing or that it has successfully completed its progress and the minimum required new replicas are available (see the `Reason` of the condition for the particulars - in our case `Reason>NewReplicaSetAvailable` means that the Deployment is complete).

You can check if a Deployment has failed to progress by using `kubectl rollout status`. `kubectl rollout status` returns a non-zero exit code if the Deployment has exceeded the progression deadline.

```
kubectl rollout status deployment/nginx-deployment
```

The output is similar to this:

```
Waiting for rollout to finish: 2 out of 3 new replicas have been updated...
```

```
error: deployment "nginx" exceeded its progress deadline
```

and the exit status from `kubectl rollout` is 1 (indicating an error):

```
echo $?
```

```
1
```

Operating on a failed deployment

All actions that apply to a complete Deployment also apply to a failed Deployment. You can scale it up/down, roll back to a previous revision, or even pause it if you need to apply multiple tweaks in the Deployment Pod template.

Clean up Policy

You can set `.spec.revisionHistoryLimit` field in a Deployment to specify how many old ReplicaSets for this Deployment you want to retain. The rest will be garbage-collected in the background. By default, it is 10.

Note: Explicitly setting this field to 0, will result in cleaning up all the history of your Deployment thus that Deployment will not be able to roll back.

Canary Deployment

If you want to roll out releases to a subset of users or servers using the Deployment, you can create multiple Deployments, one for each release, following the canary pattern described in [managing resources](#).

Writing a Deployment Spec

As with all other Kubernetes configs, a Deployment needs `.apiVersion`, `.kind`, and `.metadata` fields. For general information about working with config files, see [deploying applications](#), configuring containers, and [using kubectl to manage resources](#) documents. The name of a Deployment object must be a valid [DNS subdomain name](#).

A Deployment also needs a [.spec section](#).

Pod Template

The `.spec.template` and `.spec.selector` are the only required field of the `.spec`.

The `.spec.template` is a [Pod template](#). It has exactly the same schema as a [Pod](#), except it is nested and does not have an `apiVersion` or `kind`.

In addition to required fields for a Pod, a Pod template in a Deployment must specify appropriate labels and an appropriate restart policy. For labels, make sure not to overlap with other controllers. See [selector](#).

Only a `.spec.template.spec.restartPolicy` equal to `Always` is allowed, which is the default if not specified.

Replicas

`.spec.replicas` is an optional field that specifies the number of desired Pods. It defaults to 1.

Selector

`.spec.selector` is a required field that specifies a [label selector](#) for the Pods targeted by this Deployment.

`.spec.selector` must match `.spec.template.metadata.labels`, or it will be rejected by the API.

In API version `apps/v1`, `.spec.selector` and `.metadata.labels` do not default to `.spec.template.metadata.labels` if not set. So they must be set explicitly. Also note that `.spec.selector` is immutable after creation of the Deployment in `apps/v1`.

A Deployment may terminate Pods whose labels match the selector if their template is different from `.spec.template` or if the total number of such Pods exceeds `.spec.replicas`. It brings up new Pods with `.spec.template` if the number of Pods is less than the desired number.

Note: You should not create other Pods whose labels match this selector, either directly, by creating another Deployment, or by creating another controller such as a ReplicaSet or a ReplicationController. If you do so, the first Deployment thinks

that it created these other Pods. Kubernetes does not stop you from doing this.

If you have multiple controllers that have overlapping selectors, the controllers will fight with each other and won't behave correctly.

Strategy

.spec.strategy specifies the strategy used to replace old Pods by new ones. .spec.strategy.type can be "Recreate" or "RollingUpdate". "RollingUpdate" is the default value.

Recreate Deployment

All existing Pods are killed before new ones are created when .spec.strategy.type==Recreate.

Note: This will only guarantee Pod termination previous to creation for upgrades. If you upgrade a Deployment, all Pods of the old revision will be terminated immediately. Successful removal is awaited before any Pod of the new revision is created. If you manually delete a Pod, the lifecycle is controlled by the ReplicaSet and the replacement will be created immediately (even if the old Pod is still in a Terminating state). If you need an "at most" guarantee for your Pods, you should consider using a [StatefulSet](#).

Rolling Update Deployment

The Deployment updates Pods in a rolling update fashion when .spec.strategy.type==RollingUpdate. You can specify maxUnavailable and maxSurge to control the rolling update process.

Max Unavailable

.spec.strategy.rollingUpdate.maxUnavailable is an optional field that specifies the maximum number of Pods that can be unavailable during the update process. The value can be an absolute number (for example, 5) or a percentage of desired Pods (for example, 10%). The absolute number is calculated from percentage by rounding down. The value cannot be 0 if .spec.strategy.rollingUpdate.maxSurge is 0. The default value is 25%.

For example, when this value is set to 30%, the old ReplicaSet can be scaled down to 70% of desired Pods immediately when the rolling update starts. Once new Pods are ready, old ReplicaSet can be scaled down further, followed by scaling up the new ReplicaSet, ensuring that the total number of Pods available at all times during the update is at least 70% of the desired Pods.

Max Surge

`.spec.strategy.rollingUpdate.maxSurge` is an optional field that specifies the maximum number of Pods that can be created over the desired number of Pods. The value can be an absolute number (for example, 5) or a percentage of desired Pods (for example, 10%). The value cannot be 0 if `MaxUnavailable` is 0. The absolute number is calculated from the percentage by rounding up. The default value is 25%.

For example, when this value is set to 30%, the new ReplicaSet can be scaled up immediately when the rolling update starts, such that the total number of old and new Pods does not exceed 130% of desired Pods. Once old Pods have been killed, the new ReplicaSet can be scaled up further, ensuring that the total number of Pods running at any time during the update is at most 130% of desired Pods.

Progress Deadline Seconds

`.spec.progressDeadlineSeconds` is an optional field that specifies the number of seconds you want to wait for your Deployment to progress before the system reports back that the Deployment has [failed progressing](#) - surfaced as a condition with `Type=Progressing`, `Status=False`, and `Reason=ProgressDeadlineExceeded` in the status of the resource. The Deployment controller will keep retrying the Deployment. This defaults to 600. In the future, once automatic rollback will be implemented, the Deployment controller will roll back a Deployment as soon as it observes such a condition.

If specified, this field needs to be greater than `.spec.minReadySeconds`.

Min Ready Seconds

`.spec.minReadySeconds` is an optional field that specifies the minimum number of seconds for which a newly created Pod should be ready without any of its containers crashing, for it to be considered available. This defaults to 0 (the Pod will be considered available as soon as it is ready). To learn more about when a Pod is considered ready, see [Container Probes](#).

Revision History Limit

A Deployment's revision history is stored in the ReplicaSets it controls.

`.spec.revisionHistoryLimit` is an optional field that specifies the number of old ReplicaSets to retain to allow rollback. These old ReplicaSets consume resources in etcd and crowd the output of `kubectl get rs`. The configuration of each Deployment revision is stored in its ReplicaSets; therefore, once an old ReplicaSet is deleted, you lose the ability to rollback to that revision of Deployment. By default, 10 old ReplicaSets will be kept, however its ideal value depends on the frequency and stability of new Deployments.

More specifically, setting this field to zero means that all old ReplicaSets with 0 replicas will be cleaned up. In this case, a new Deployment rollout cannot be undone, since its revision history is cleaned up.

Paused

`.spec.paused` is an optional boolean field for pausing and resuming a Deployment. The only difference between a paused Deployment and one that is not paused, is that any changes into the PodTemplateSpec of the paused Deployment will not trigger new rollouts as long as it is paused. A Deployment is not paused by default when it is created.

Feedback

Was this page helpful?

Yes No

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

Last modified December 08, 2020 at 6:29 PM PST: [Update rollout deployment command \(fd411a39c\)](#)

[Edit this page](#) [Create child page](#) [Create an issue](#)

- [Use Case](#)
- [Creating a Deployment](#)
 - [Pod-template-hash label](#)
- [Updating a Deployment](#)
 - [Rollover \(aka multiple updates in-flight\)](#)
 - [Label selector updates](#)
- [Rolling Back a Deployment](#)
 - [Checking Rollout History of a Deployment](#)
 - [Rolling Back to a Previous Revision](#)
- [Scaling a Deployment](#)
 - [Proportional scaling](#)
- [Pausing and Resuming a Deployment](#)
- [Deployment status](#)
 - [Progressing Deployment](#)
 - [Complete Deployment](#)
 - [Failed Deployment](#)
 - [Operating on a failed deployment](#)
- [Clean up Policy](#)
- [Canary Deployment](#)
- [Writing a Deployment Spec](#)
 - [Pod Template](#)
 - [Replicas](#)
 - [Selector](#)
 - [Strategy](#)
 - [Progress Deadline Seconds](#)

- [Min Ready Seconds](#)
- [Revision History Limit](#)
- [Paused](#)

ReplicaSet

A ReplicaSet's purpose is to maintain a stable set of replica Pods running at any given time. As such, it is often used to guarantee the availability of a specified number of identical Pods.

How a ReplicaSet works

A ReplicaSet is defined with fields, including a selector that specifies how to identify Pods it can acquire, a number of replicas indicating how many Pods it should be maintaining, and a pod template specifying the data of new Pods it should create to meet the number of replicas criteria. A ReplicaSet then fulfills its purpose by creating and deleting Pods as needed to reach the desired number. When a ReplicaSet needs to create new Pods, it uses its Pod template.

A ReplicaSet is linked to its Pods via the Pods' [metadata.ownerReferences](#) field, which specifies what resource the current object is owned by. All Pods acquired by a ReplicaSet have their owning ReplicaSet's identifying information within their ownerReferences field. It's through this link that the ReplicaSet knows of the state of the Pods it is maintaining and plans accordingly.

A ReplicaSet identifies new Pods to acquire by using its selector. If there is a Pod that has no OwnerReference or the OwnerReference is not a [Controller](#) and it matches a ReplicaSet's selector, it will be immediately acquired by said ReplicaSet.

When to use a ReplicaSet

A ReplicaSet ensures that a specified number of pod replicas are running at any given time. However, a Deployment is a higher-level concept that manages ReplicaSets and provides declarative updates to Pods along with a lot of other useful features. Therefore, we recommend using Deployments instead of directly using ReplicaSets, unless you require custom update orchestration or don't require updates at all.

This actually means that you may never need to manipulate ReplicaSet objects: use a Deployment instead, and define your application in the spec section.

Example

[controllers/frontend.yaml](#)



```
apiVersion: apps/v1
kind: ReplicaSet
metadata:
  name: frontend
  labels:
    app: guestbook
    tier: frontend
spec:
  # modify replicas according to your case
  replicas: 3
  selector:
    matchLabels:
      tier: frontend
  template:
    metadata:
      labels:
        tier: frontend
    spec:
      containers:
        - name: php-redis
          image: gcr.io/google_samples/gb-frontend:v3
```

Saving this manifest into `frontend.yaml` and submitting it to a Kubernetes cluster will create the defined `ReplicaSet` and the Pods that it manages.

```
kubectl apply -f https://kubernetes.io/examples/controllers/
frontend.yaml
```

You can then get the current `ReplicaSets` deployed:

```
kubectl get rs
```

And see the frontend one you created:

NAME	DESIRED	CURRENT	READY	AGE
frontend	3	3	3	6s

You can also check on the state of the `ReplicaSet`:

```
kubectl describe rs/frontend
```

And you will see output similar to:

Name:	frontend
Namespace:	default
Selector:	<code>tier=frontend</code>
Labels:	<code>app=guestbook</code>

```

tier=frontend
Annotations: kubectl.kubernetes.io/last-applied-configuration:
              {"apiVersion":"apps/v1","kind":"ReplicaSet","meta
data":{"annotations":{},"labels":{"app":"guestbook","tier":"front
end"},"name":"frontend",...}
Replicas: 3 current / 3 desired
Pods Status: 3 Running / 0 Waiting / 0 Succeeded / 0 Failed
Pod Template:
Labels: tier=frontend
Containers:
php-redis:
Image: gcr.io/google_samples/gb-frontend:v3
Port: <none>
Host Port: <none>
Environment: <none>
Mounts: <none>
Volumes: <none>
Events:
Type Reason Age From Message
---- ---- - - -
Normal SuccessfulCreate 117s replicaset-controller Created
pod: frontend-wtsmm
Normal SuccessfulCreate 116s replicaset-controller Created
pod: frontend-b2zdv
Normal SuccessfulCreate 116s replicaset-controller Created
pod: frontend-vcmts

```

And lastly you can check for the Pods brought up:

```
kubectl get pods
```

You should see Pod information similar to:

NAME	READY	STATUS	RESTARTS	AGE
frontend-b2zdv	1/1	Running	0	6m36s
frontend-vcmts	1/1	Running	0	6m36s
frontend-wtsmm	1/1	Running	0	6m36s

You can also verify that the owner reference of these pods is set to the frontend ReplicaSet. To do this, get the yaml of one of the Pods running:

```
kubectl get pods frontend-b2zdv -o yaml
```

The output will look similar to this, with the frontend ReplicaSet's info set in the metadata's ownerReferences field:

```

apiVersion: v1
kind: Pod
metadata:
  creationTimestamp: "2020-02-12T07:06:16Z"
  generateName: frontend-
  labels:
    tier: frontend

```

```
name: frontend-b2zdv
namespace: default
ownerReferences:
- apiVersion: apps/v1
  blockOwnerDeletion: true
  controller: true
  kind: ReplicaSet
  name: frontend
  uid: f391f6db-bb9b-4c09-ae74-6a1f77f3d5cf
...
```

Non-Template Pod acquisitions

While you can create bare Pods with no problems, it is strongly recommended to make sure that the bare Pods do not have labels which match the selector of one of your ReplicaSets. The reason for this is because a ReplicaSet is not limited to owning Pods specified by its template-- it can acquire other Pods in the manner specified in the previous sections.

Take the previous frontend ReplicaSet example, and the Pods specified in the following manifest:

[pods/pod-rs.yaml](#)



```
apiVersion: v1
kind: Pod
metadata:
  name: pod1
  labels:
    tier: frontend
spec:
  containers:
  - name: hello1
    image: gcr.io/google-samples/hello-app:2.0
---
apiVersion: v1
kind: Pod
metadata:
  name: pod2
  labels:
    tier: frontend
spec:
  containers:
  - name: hello2
    image: gcr.io/google-samples/hello-app:1.0
```

As those Pods do not have a Controller (or any object) as their owner reference and match the selector of the frontend ReplicaSet, they will immediately be acquired by it.

Suppose you create the Pods after the frontend ReplicaSet has been deployed and has set up its initial Pod replicas to fulfill its replica count requirement:

```
kubectl apply -f https://kubernetes.io/examples/pods/pod-rs.yaml
```

The new Pods will be acquired by the ReplicaSet, and then immediately terminated as the ReplicaSet would be over its desired count.

Fetching the Pods:

```
kubectl get pods
```

The output shows that the new Pods are either already terminated, or in the process of being terminated:

NAME	READY	STATUS	RESTARTS	AGE
frontend-b2zdv	1/1	Running	0	10m
frontend-vcmts	1/1	Running	0	10m
frontend-wtsmm	1/1	Running	0	10m
pod1	0/1	Terminating	0	1s
pod2	0/1	Terminating	0	1s

If you create the Pods first:

```
kubectl apply -f https://kubernetes.io/examples/pods/pod-rs.yaml
```

And then create the ReplicaSet however:

```
kubectl apply -f https://kubernetes.io/examples/controllers/frontend.yaml
```

You shall see that the ReplicaSet has acquired the Pods and has only created new ones according to its spec until the number of its new Pods and the original matches its desired count. As fetching the Pods:

```
kubectl get pods
```

Will reveal in its output:

NAME	READY	STATUS	RESTARTS	AGE
frontend-hmmj2	1/1	Running	0	9s
pod1	1/1	Running	0	36s
pod2	1/1	Running	0	36s

In this manner, a ReplicaSet can own a non-homogenous set of Pods

Writing a ReplicaSet manifest

As with all other Kubernetes API objects, a ReplicaSet needs the `apiVersion`, `kind`, and `metadata` fields. For ReplicaSets, the `kind` is always just `ReplicaSet`. In Kubernetes 1.9 the API version `apps/v1` on the `ReplicaSet` kind is the current version and is enabled by default. The API version `apps/v1beta2` is deprecated. Refer to the first lines of the `frontend.yaml` example for guidance.

The name of a ReplicaSet object must be a valid [DNS subdomain name](#).

A ReplicaSet also needs a [.spec section](#).

Pod Template

The `.spec.template` is a [pod template](#) which is also required to have labels in place. In our `frontend.yaml` example we had one label: `tier: frontend`. Be careful not to overlap with the selectors of other controllers, lest they try to adopt this Pod.

For the template's [restart policy](#) field, `.spec.template.spec.restartPolicy`, the only allowed value is `Always`, which is the default.

Pod Selector

The `.spec.selector` field is a [label selector](#). As discussed [earlier](#) these are the labels used to identify potential Pods to acquire. In our `frontend.yaml` example, the selector was:

```
matchLabels:  
  tier: frontend
```

In the ReplicaSet, `.spec.template.metadata.labels` must match `spec.selector`, or it will be rejected by the API.

Note: For 2 ReplicaSets specifying the same `.spec.selector` but different `.spec.template.metadata.labels` and `.spec.template.spec` fields, each ReplicaSet ignores the Pods created by the other ReplicaSet.

Replicas

You can specify how many Pods should run concurrently by setting `.spec.replicas`. The ReplicaSet will create/delete its Pods to match this number.

If you do not specify `.spec.replicas`, then it defaults to 1.

Working with ReplicaSets

Deleting a ReplicaSet and its Pods

To delete a ReplicaSet and all of its Pods, use [kubectl delete](#). The [Garbage collector](#) automatically deletes all of the dependent Pods by default.

When using the REST API or the `client-go` library, you must set `propagationPolicy` to `Background` or `Foreground` in the `-d` option. For example:

```
kubectl proxy --port=8080
curl -X DELETE 'localhost:8080/apis/apps/v1/namespaces/default/
replicasets/frontend' \
> -d '{"kind":"DeleteOptions","apiVersion":"v1","propagationPolicy":"Foreground"}' \
> -H "Content-Type: application/json"
```

Deleting just a ReplicaSet

You can delete a ReplicaSet without affecting any of its Pods using [kubectl delete](#) with the `--cascade=false` option. When using the REST API or the `client-go` library, you must set `propagationPolicy` to `Orphan`. For example:

```
kubectl proxy --port=8080
curl -X DELETE 'localhost:8080/apis/apps/v1/namespaces/default/
replicasets/frontend' \
> -d '{"kind":"DeleteOptions","apiVersion":"v1","propagationPolicy":"Orphan"}' \
> -H "Content-Type: application/json"
```

Once the original is deleted, you can create a new ReplicaSet to replace it. As long as the old and new `.spec.selector` are the same, then the new one will adopt the old Pods. However, it will not make any effort to make existing Pods match a new, different pod template. To update Pods to a new spec in a controlled way, use a [Deployment](#), as ReplicaSets do not support a rolling update directly.

Isolating Pods from a ReplicaSet

You can remove Pods from a ReplicaSet by changing their labels. This technique may be used to remove Pods from service for debugging, data recovery, etc. Pods that are removed in this way will be replaced automatically (assuming that the number of replicas is not also changed).

Scaling a ReplicaSet

A ReplicaSet can be easily scaled up or down by simply updating the `.spec.replicas` field. The ReplicaSet controller ensures that a desired number of Pods with a matching label selector are available and operational.

ReplicaSet as a Horizontal Pod Autoscaler Target

A ReplicaSet can also be a target for [Horizontal Pod Autoscalers \(HPA\)](#). That is, a ReplicaSet can be auto-scaled by an HPA. Here is an example HPA targeting the ReplicaSet we created in the previous example.

[controllers/hpa-rs.yaml](#)



```
apiVersion: autoscaling/v1
kind: HorizontalPodAutoscaler
metadata:
  name: frontend-scaler
spec:
  scaleTargetRef:
    kind: ReplicaSet
    name: frontend
  minReplicas: 3
  maxReplicas: 10
  targetCPUUtilizationPercentage: 50
```

Saving this manifest into `hpa-rs.yaml` and submitting it to a Kubernetes cluster should create the defined HPA that autoscales the target ReplicaSet depending on the CPU usage of the replicated Pods.

```
kubectl apply -f https://k8s.io/examples/controllers/hpa-rs.yaml
```

Alternatively, you can use the `kubectl autoscale` command to accomplish the same (and it's easier!)

```
kubectl autoscale rs frontend --max=10 --min=3 --cpu-percent=50
```

Alternatives to ReplicaSet

Deployment (recommended)

[Deployment](#) is an object which can own ReplicaSets and update them and their Pods via declarative, server-side rolling updates. While ReplicaSets can be used independently, today they're mainly used by Deployments as a mechanism to orchestrate Pod creation, deletion and updates. When you use Deployments you don't have to worry about managing the ReplicaSets that they create. Deployments own and manage their ReplicaSets. As such, it is recommended to use Deployments when you want ReplicaSets.

Bare Pods

Unlike the case where a user directly created Pods, a ReplicaSet replaces Pods that are deleted or terminated for any reason, such as in the case of node failure or disruptive node maintenance, such as a kernel upgrade. For this reason, we recommend that you use a ReplicaSet even if your application requires only a single Pod. Think of it similarly to a process

supervisor, only it supervises multiple Pods across multiple nodes instead of individual processes on a single node. A ReplicaSet delegates local container restarts to some agent on the node (for example, Kubelet or Docker).

Job

Use a [Job](#) instead of a ReplicaSet for Pods that are expected to terminate on their own (that is, batch jobs).

DaemonSet

Use a [DaemonSet](#) instead of a ReplicaSet for Pods that provide a machine-level function, such as machine monitoring or machine logging. These Pods have a lifetime that is tied to a machine lifetime: the Pod needs to be running on the machine before other Pods start, and are safe to terminate when the machine is otherwise ready to be rebooted/shutdown.

ReplicationController

ReplicaSets are the successors to [ReplicationControllers](#). The two serve the same purpose, and behave similarly, except that a ReplicationController does not support set-based selector requirements as described in the [labels user guide](#). As such, ReplicaSets are preferred over ReplicationControllers

Feedback

Was this page helpful?

Yes *No*

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

Last modified October 22, 2020 at 2:24 PM PST: [Fix links in concepts section \(070023b24\)](#)

[Edit this page](#) [Create child page](#) [Create an issue](#)

- [How a ReplicaSet works](#)
- [When to use a ReplicaSet](#)
- [Example](#)
- [Non-Template Pod acquisitions](#)
- [Writing a ReplicaSet manifest](#)
 - [Pod Template](#)
 - [Pod Selector](#)
 - [Replicas](#)
- [Working with ReplicaSets](#)
 - [Deleting a ReplicaSet and its Pods](#)
 - [Deleting just a ReplicaSet](#)
 - [Isolating Pods from a ReplicaSet](#)
 - [Scaling a ReplicaSet](#)

- [ReplicaSet as a Horizontal Pod Autoscaler Target](#)
- [Alternatives to ReplicaSet](#)
 - [Deployment \(recommended\)](#)
 - [Bare Pods](#)
 - [Job](#)
 - [DaemonSet](#)
 - [ReplicationController](#)

StatefulSets

StatefulSet is the workload API object used to manage stateful applications.

Manages the deployment and scaling of a set of [Pods](#), and provides guarantees about the ordering and uniqueness of these Pods.

Like a [Deployment](#), a StatefulSet manages Pods that are based on an identical container spec. Unlike a Deployment, a StatefulSet maintains a sticky identity for each of their Pods. These pods are created from the same spec, but are not interchangeable: each has a persistent identifier that it maintains across any rescheduling.

If you want to use storage volumes to provide persistence for your workload, you can use a StatefulSet as part of the solution. Although individual Pods in a StatefulSet are susceptible to failure, the persistent Pod identifiers make it easier to match existing volumes to the new Pods that replace any that have failed.

Using StatefulSets

StatefulSets are valuable for applications that require one or more of the following.

- *Stable, unique network identifiers.*
- *Stable, persistent storage.*
- *Ordered, graceful deployment and scaling.*
- *Ordered, automated rolling updates.*

In the above, stable is synonymous with persistence across Pod (re)scheduling. If an application doesn't require any stable identifiers or ordered deployment, deletion, or scaling, you should deploy your application using a workload object that provides a set of stateless replicas. [Deployment](#) or [ReplicaSet](#) may be better suited to your stateless needs.

Limitations

- *The storage for a given Pod must either be provisioned by a [PersistentVolume Provisioner](#) based on the requested storage class, or pre-provisioned by an admin.*
- *Deleting and/or scaling a StatefulSet down will not delete the volumes associated with the StatefulSet. This is done to ensure data safety,*

which is generally more valuable than an automatic purge of all related StatefulSet resources.

- *StatefulSets currently require a [Headless Service](#) to be responsible for the network identity of the Pods. You are responsible for creating this Service.*
- *StatefulSets do not provide any guarantees on the termination of pods when a StatefulSet is deleted. To achieve ordered and graceful termination of the pods in the StatefulSet, it is possible to scale the StatefulSet down to 0 prior to deletion.*
- *When using [Rolling Updates](#) with the default [Pod Management Policy](#) (0 rderedReady), it's possible to get into a broken state that requires [manual intervention to repair](#).*

Components

The example below demonstrates the components of a StatefulSet.

```
apiVersion: v1
kind: Service
metadata:
  name: nginx
  labels:
    app: nginx
spec:
  ports:
  - port: 80
    name: web
  clusterIP: None
  selector:
    app: nginx
---
apiVersion: apps/v1
kind: StatefulSet
metadata:
  name: web
spec:
  selector:
    matchLabels:
      app: nginx # has to match .spec.template.metadata.labels
  serviceName: "nginx"
  replicas: 3 # by default is 1
  template:
    metadata:
      labels:
        app: nginx # has to match .spec.selector.matchLabels
  spec:
    terminationGracePeriodSeconds: 10
    containers:
    - name: nginx
      image: k8s.gcr.io/nginx-slim:0.8
      ports:
```

```

    - containerPort: 80
      name: web
    volumeMounts:
    - name: www
      mountPath: /usr/share/nginx/html
volumeClaimTemplates:
- metadata:
  name: www
spec:
  accessModes: [ "ReadWriteOnce" ]
  storageClassName: "my-storage-class"
  resources:
  requests:
    storage: 1Gi

```

In the above example:

- A Headless Service, named `nginx`, is used to control the network domain.
- The StatefulSet, named `web`, has a Spec that indicates that 3 replicas of the `nginx` container will be launched in unique Pods.
- The `volumeClaimTemplates` will provide stable storage using [PersistentVolumes](#) provisioned by a PersistentVolume Provisioner.

The name of a StatefulSet object must be a valid [DNS subdomain name](#).

Pod Selector

You must set the `.spec.selector` field of a StatefulSet to match the labels of its `.spec.template.metadata.labels`. Prior to Kubernetes 1.8, the `.spec.selector` field was defaulted when omitted. In 1.8 and later versions, failing to specify a matching Pod Selector will result in a validation error during StatefulSet creation.

Pod Identity

StatefulSet Pods have a unique identity that is comprised of an ordinal, a stable network identity, and stable storage. The identity sticks to the Pod, regardless of which node it's (re)scheduled on.

Ordinal Index

For a StatefulSet with N replicas, each Pod in the StatefulSet will be assigned an integer ordinal, from 0 up through $N-1$, that is unique over the Set.

Stable Network ID

Each Pod in a StatefulSet derives its hostname from the name of the StatefulSet and the ordinal of the Pod. The pattern for the constructed

hostname is \$(statefulset name)-\$(ordinal). The example above will create three Pods named web-0, web-1, web-2. A StatefulSet can use a [Headless Service](#) to control the domain of its Pods. The domain managed by this Service takes the form: \$(service name).\$(namespace).svc.cluster.local, where "cluster.local" is the cluster domain. As each Pod is created, it gets a matching DNS subdomain, taking the form: \$(podname).\$(governing service domain), where the governing service is defined by the serviceName field on the StatefulSet.

Depending on how DNS is configured in your cluster, you may not be able to look up the DNS name for a newly-run Pod immediately. This behavior can occur when other clients in the cluster have already sent queries for the hostname of the Pod before it was created. Negative caching (normal in DNS) means that the results of previous failed lookups are remembered and reused, even after the Pod is running, for at least a few seconds.

If you need to discover Pods promptly after they are created, you have a few options:

- Query the Kubernetes API directly (for example, using a watch) rather than relying on DNS lookups.
- Decrease the time of caching in your Kubernetes DNS provider (typically this means editing the config map for CoreDNS, which currently caches for 30 seconds).

As mentioned in the [limitations](#) section, you are responsible for creating the [Headless Service](#) responsible for the network identity of the pods.

Here are some examples of choices for Cluster Domain, Service name, StatefulSet name, and how that affects the DNS names for the StatefulSet's Pods.

Cluster Domain	Service (ns/name)	StatefulSet (ns/name)	StatefulSet Domain	Pod DNS
cluster.local	default/nginx	default/web	nginx.default.svc.cluster.local	web-{0..N-1}.nginx.default.svc.cluster.local
cluster.local	foo/nginx	foo/web	nginx.foo.svc.cluster.local	web-{0..N-1}.nginx.foo.svc.cluster.local
kube.local	foo/nginx	foo/web	nginx.foo.svc.kube.local	web-{0..N-1}.nginx.foo.svc.kube.local

Note: Cluster Domain will be set to `cluster.local` unless [otherwise configured](#).

Stable Storage

Kubernetes creates one [PersistentVolume](#) for each `VolumeClaimTemplate`. In the nginx example above, each Pod will receive a single PersistentVolume with a `StorageClass` of `my-storage-class` and 1 Gib of provisioned storage. If no `StorageClass` is specified, then the default `StorageClass` will be used. When a Pod is (re)scheduled onto a node, its `volumeMounts` mount the

PersistentVolumes associated with its PersistentVolume Claims. Note that, the PersistentVolumes associated with the Pods' PersistentVolume Claims are not deleted when the Pods, or StatefulSet are deleted. This must be done manually.

Pod Name Label

When the StatefulSet [Controller](#) creates a Pod, it adds a label, `statefulset.kubernetes.io/pod-name`, that is set to the name of the Pod. This label allows you to attach a Service to a specific Pod in the StatefulSet.

Deployment and Scaling Guarantees

- For a StatefulSet with N replicas, when Pods are being deployed, they are created sequentially, in order from $\{0..N-1\}$.
- When Pods are being deleted, they are terminated in reverse order, from $\{N-1..0\}$.
- Before a scaling operation is applied to a Pod, all of its predecessors must be Running and Ready.
- Before a Pod is terminated, all of its successors must be completely shutdown.

The StatefulSet should not specify a `pod.Spec.TerminationGracePeriodSeconds` of 0. This practice is unsafe and strongly discouraged. For further explanation, please refer to [force deleting StatefulSet Pods](#).

When the nginx example above is created, three Pods will be deployed in the order web-0, web-1, web-2. web-1 will not be deployed before web-0 is [Running and Ready](#), and web-2 will not be deployed until web-1 is Running and Ready. If web-0 should fail, after web-1 is Running and Ready, but before web-2 is launched, web-2 will not be launched until web-0 is successfully relaunched and becomes Running and Ready.

If a user were to scale the deployed example by patching the StatefulSet such that `replicas=1`, web-2 would be terminated first. web-1 would not be terminated until web-2 is fully shutdown and deleted. If web-0 were to fail after web-2 has been terminated and is completely shutdown, but prior to web-1's termination, web-1 would not be terminated until web-0 is Running and Ready.

Pod Management Policies

In Kubernetes 1.7 and later, StatefulSet allows you to relax its ordering guarantees while preserving its uniqueness and identity guarantees via its `.spec.podManagementPolicy` field.

OrderedReady Pod Management

OrderedReady pod management is the default for StatefulSets. It implements the behavior described [above](#).

Parallel Pod Management

Parallel pod management tells the StatefulSet controller to launch or terminate all Pods in parallel, and to not wait for Pods to become Running and Ready or completely terminated prior to launching or terminating another Pod. This option only affects the behavior for scaling operations. Updates are not affected.

Update Strategies

In Kubernetes 1.7 and later, StatefulSet's .spec.updateStrategy field allows you to configure and disable automated rolling updates for containers, labels, resource request/limits, and annotations for the Pods in a StatefulSet.

On Delete

The OnDelete update strategy implements the legacy (1.6 and prior) behavior. When a StatefulSet's .spec.updateStrategy.type is set to OnDelete, the StatefulSet controller will not automatically update the Pods in a StatefulSet. Users must manually delete Pods to cause the controller to create new Pods that reflect modifications made to a StatefulSet's .spec.template.

Rolling Updates

The RollingUpdate update strategy implements automated, rolling update for the Pods in a StatefulSet. It is the default strategy when .spec.updateStrategy is left unspecified. When a StatefulSet's .spec.updateStrategy.type is set to RollingUpdate, the StatefulSet controller will delete and recreate each Pod in the StatefulSet. It will proceed in the same order as Pod termination (from the largest ordinal to the smallest), updating each Pod one at a time. It will wait until an updated Pod is Running and Ready prior to updating its predecessor.

Partitions

The RollingUpdate update strategy can be partitioned, by specifying a .spec.updateStrategy.rollingUpdate.partition. If a partition is specified, all Pods with an ordinal that is greater than or equal to the partition will be updated when the StatefulSet's .spec.template is updated. All Pods with an ordinal that is less than the partition will not be updated, and, even if they are deleted, they will be recreated at the previous version. If a StatefulSet's .spec.updateStrategy.rollingUpdate.partition is greater than its .spec.replicas, updates to its .spec.template will not be propagated to its Pods. In most cases you will not need to use a partition, but they are useful if you want to stage an update, roll out a canary, or perform a phased roll out.

Forced Rollback

When using [Rolling Updates](#) with the default [Pod Management Policy](#) (`OrderedReady`), it's possible to get into a broken state that requires manual intervention to repair.

If you update the Pod template to a configuration that never becomes `Running` and `Ready` (for example, due to a bad binary or application-level configuration error), `StatefulSet` will stop the rollout and wait.

In this state, it's not enough to revert the Pod template to a good configuration. Due to a [known issue](#), `StatefulSet` will continue to wait for the broken Pod to become `Ready` (which never happens) before it will attempt to revert it back to the working configuration.

After reverting the template, you must also delete any Pods that `StatefulSet` had already attempted to run with the bad configuration. `StatefulSet` will then begin to recreate the Pods using the reverted template.

What's next

- Follow an example of [deploying a stateful application](#).
- Follow an example of [deploying Cassandra with Stateful Sets](#).
- Follow an example of [running a replicated stateful application](#).

Feedback

Was this page helpful?

Yes No

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

Last modified October 19, 2020 at 1:29 PM PST: [Fix minor typo in StatefulSets docs \(427c96e64\)](#)

[Edit this page](#) [Create child page](#) [Create an issue](#)

- [Using StatefulSets](#)
- [Limitations](#)
- [Components](#)
- [Pod Selector](#)
- [Pod Identity](#)
 - [Ordinal Index](#)
 - [Stable Network ID](#)
 - [Stable Storage](#)
 - [Pod Name Label](#)
- [Deployment and Scaling Guarantees](#)
 - [Pod Management Policies](#)

- [Update Strategies](#)
 - [On Delete](#)
 - [Rolling Updates](#)
- [What's next](#)

DaemonSet

A DaemonSet ensures that all (or some) Nodes run a copy of a Pod. As nodes are added to the cluster, Pods are added to them. As nodes are removed from the cluster, those Pods are garbage collected. Deleting a DaemonSet will clean up the Pods it created.

Some typical uses of a DaemonSet are:

- running a cluster storage daemon on every node
- running a logs collection daemon on every node
- running a node monitoring daemon on every node

In a simple case, one DaemonSet, covering all nodes, would be used for each type of daemon. A more complex setup might use multiple DaemonSets for a single type of daemon, but with different flags and/or different memory and cpu requests for different hardware types.

Writing a DaemonSet Spec

Create a DaemonSet

You can describe a DaemonSet in a YAML file. For example, the `daemonset.yaml` file below describes a DaemonSet that runs the fluentd-elasticsearch Docker image:

[controllers/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
  template:
    metadata:
      labels:
        name: fluentd-elasticsearch
    spec:
```

```

tolerations:
# this toleration is to have the daemonset runnable on
master nodes
- key: node-role.kubernetes.io/master
  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

```

Create a DaemonSet based on the YAML file:

```
kubectl apply -f https://k8s.io/examples/controllers/
daemonset.yaml
```

Required Fields

As with all other Kubernetes config, a DaemonSet needs `apiVersion`, `kind`, and `metadata` fields. For general information about working with config files, see [running stateless applications](#), [configuring containers](#), and [object management using kubectl](#) documents.

The name of a DaemonSet object must be a valid [DNS subdomain name](#).

A DaemonSet also needs a [.spec](#) section.

Pod Template

The `.spec.template` is one of the required fields in `.spec`.

The `.spec.template` is a [pod template](#). It has exactly the same schema as a [Pod](#), except it is nested and does not have an `apiVersion` or `kind`.

In addition to required fields for a Pod, a Pod template in a DaemonSet has to specify appropriate labels (see [pod selector](#)).

A Pod Template in a DaemonSet must have a [RestartPolicy](#) equal to Always, or be unspecified, which defaults to Always.

Pod Selector

The .spec.selector field is a pod selector. It works the same as the .spec.selector of a [Job](#).

As of Kubernetes 1.8, you must specify a pod selector that matches the labels of the .spec.template. The pod selector will no longer be defaulted when left empty. Selector defaulting was not compatible with kubectl apply. Also, once a DaemonSet is created, its .spec.selector can not be mutated. Mutating the pod selector can lead to the unintentional orphaning of Pods, and it was found to be confusing to users.

The .spec.selector is an object consisting of two fields:

- *matchLabels - works the same as the .spec.selector of a [ReplicationController](#).*
- *matchExpressions - allows to build more sophisticated selectors by specifying key, list of values and an operator that relates the key and values.*

When the two are specified the result is ANDed.

If the .spec.selector is specified, it must match the .spec.template.metadata.labels. Config with these not matching will be rejected by the API.

Running Pods on select Nodes

If you specify a .spec.template.spec.nodeName, then the DaemonSet controller will create Pods on nodes which match that [node selector](#).

Likewise if you specify a .spec.template.spec.affinity, then DaemonSet controller will create Pods on nodes which match that [node affinity](#). If you do not specify either, then the DaemonSet controller will create Pods on all nodes.

How Daemon Pods are scheduled

Scheduled by default scheduler

FEATURE STATE: Kubernetes v1.20 [stable]

A DaemonSet ensures that all eligible nodes run a copy of a Pod. Normally, the node that a Pod runs on is selected by the Kubernetes scheduler.

However, DaemonSet pods are created and scheduled by the DaemonSet controller instead. That introduces the following issues:

- Inconsistent Pod behavior: Normal Pods waiting to be scheduled are created and in Pending state, but DaemonSet pods are not created in Pending state. This is confusing to the user.
- [Pod preemption](#) is handled by default scheduler. When preemption is enabled, the DaemonSet controller will make scheduling decisions without considering pod priority and preemption.

`ScheduleDaemonSetPods` allows you to schedule DaemonSets using the default scheduler instead of the DaemonSet controller, by adding the `NodeAffinity` term to the DaemonSet pods, instead of the `.spec.nodeName` term. The default scheduler is then used to bind the pod to the target host. If node affinity of the DaemonSet pod already exists, it is replaced (the original node affinity was taken into account before selecting the target host). The DaemonSet controller only performs these operations when creating or modifying DaemonSet pods, and no changes are made to the `spec.template` of the DaemonSet.

```
nodeAffinity:  
  requiredDuringSchedulingIgnoredDuringExecution:  
    nodeSelectorTerms:  
      - matchFields:  
        - key: metadata.name  
          operator: In  
          values:  
            - target-host-name
```

In addition, `node.kubernetes.io/unschedulable:NoSchedule` toleration is added automatically to DaemonSet Pods. The default scheduler ignores unschedulable Nodes when scheduling DaemonSet Pods.

Taints and Tolerations

Although Daemon Pods respect [taints and tolerations](#), the following tolerations are added to DaemonSet Pods automatically according to the related features.

Toleration Key	Effect	Version	Description
<code>node.kubernetes.io/not-ready</code>	NoExecute	1.13+	DaemonSet pods will not be evicted when there are node problems such as a network partition.
<code>node.kubernetes.io/unreachable</code>	NoExecute	1.13+	DaemonSet pods will not be evicted when there are node problems such as a network partition.
<code>node.kubernetes.io/disk-pressure</code>	NoSchedule	1.8+	

Toleration Key	Effect	Version	Description
node.kubernetes.io/memory-pressure	NoSchedule	1.8+	
node.kubernetes.io/unschedulable	NoSchedule	1.12+	DaemonSet pods tolerate unschedulable attributes by default scheduler.
node.kubernetes.io/network-unavailable	NoSchedule	1.12+	DaemonSet pods, who uses host network, tolerate network-unavailable attributes by default scheduler.

Communicating with Daemon Pods

Some possible patterns for communicating with Pods in a DaemonSet are:

- **Push:** Pods in the DaemonSet are configured to send updates to another service, such as a stats database. They do not have clients.
- **NodeIP and Known Port:** Pods in the DaemonSet can use a `hostPort`, so that the pods are reachable via the node IPs. Clients know the list of node IPs somehow, and know the port by convention.
- **DNS:** Create a [headless service](#) with the same pod selector, and then discover DaemonSets using the `endpoints` resource or retrieve multiple A records from DNS.
- **Service:** Create a service with the same Pod selector, and use the service to reach a daemon on a random node. (No way to reach specific node.)

Updating a DaemonSet

If node labels are changed, the DaemonSet will promptly add Pods to newly matching nodes and delete Pods from newly not-matching nodes.

You can modify the Pods that a DaemonSet creates. However, Pods do not allow all fields to be updated. Also, the DaemonSet controller will use the original template the next time a node (even with the same name) is created.

You can delete a DaemonSet. If you specify `--cascade=false` with `kubectl`, then the Pods will be left on the nodes. If you subsequently create a new DaemonSet with the same selector, the new DaemonSet adopts the existing Pods. If any Pods need replacing the DaemonSet replaces them according to its `updateStrategy`.

You can [perform a rolling update](#) on a DaemonSet.

Alternatives to DaemonSet

Init scripts

It is certainly possible to run daemon processes by directly starting them on a node (e.g. using `init`, `upstartd`, or `systemd`). This is perfectly fine. However, there are several advantages to running such processes via a DaemonSet:

- Ability to monitor and manage logs for daemons in the same way as applications.
- Same config language and tools (e.g. Pod templates, `kubectl`) for daemons and applications.
- Running daemons in containers with resource limits increases isolation between daemons from app containers. However, this can also be accomplished by running the daemons in a container but not in a Pod (e.g. start directly via Docker).

Bare Pods

It is possible to create Pods directly which specify a particular node to run on. However, a DaemonSet replaces Pods that are deleted or terminated for any reason, such as in the case of node failure or disruptive node maintenance, such as a kernel upgrade. For this reason, you should use a DaemonSet rather than creating individual Pods.

Static Pods

It is possible to create Pods by writing a file to a certain directory watched by Kubelet. These are called [static pods](#). Unlike DaemonSet, static Pods cannot be managed with `kubectl` or other Kubernetes API clients. Static Pods do not depend on the apiserver, making them useful in cluster bootstrapping cases. Also, static Pods may be deprecated in the future.

Deployments

DaemonSets are similar to [Deployments](#) in that they both create Pods, and those Pods have processes which are not expected to terminate (e.g. web servers, storage servers).

Use a Deployment for stateless services, like frontends, where scaling up and down the number of replicas and rolling out updates are more important than controlling exactly which host the Pod runs on. Use a DaemonSet when it is important that a copy of a Pod always run on all or certain hosts, and when it needs to start before other Pods.

Feedback

Was this page helpful?

Yes No

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

Last modified December 03, 2020 at 5:20 PM PST: [Remove problems in DaemonSet fixed by controllerRef \(e837312f1\)](#)

[Edit this page](#) [Create child page](#) [Create an issue](#)

- [Writing a DaemonSet Spec](#)
 - [Create a DaemonSet](#)
 - [Required Fields](#)
 - [Pod Template](#)
 - [Pod Selector](#)
 - [Running Pods on select Nodes](#)
- [How Daemon Pods are scheduled](#)
 - [Scheduled by default scheduler](#)
 - [Taints and Tolerations](#)
- [Communicating with Daemon Pods](#)
- [Updating a DaemonSet](#)
- [Alternatives to DaemonSet](#)
 - [Init scripts](#)
 - [Bare Pods](#)
 - [Static Pods](#)
 - [Deployments](#)

Jobs

A Job creates one or more Pods and ensures that a specified number of them successfully terminate. As pods successfully complete, the Job tracks the successful completions. When a specified number of successful completions is reached, the task (ie, Job) is complete. Deleting a Job will clean up the Pods it created.

A simple case is to create one Job object in order to reliably run one Pod to completion. The Job object will start a new Pod if the first Pod fails or is deleted (for example due to a node hardware failure or a node reboot).

You can also use a Job to run multiple Pods in parallel.

Running an example Job

Here is an example Job config. It computes π to 2000 places and prints it out. It takes around 10s to complete.

[controllers/job.yaml](#)



```

apiVersion: batch/v1
kind: Job
metadata:
  name: pi
spec:
  template:
    spec:
      containers:
        - name: pi
          image: perl
          command: ["perl", "-Mbignum=bpi", "-wle", "print
bpi(2000)"]
        restartPolicy: Never
        backoffLimit: 4

```

You can run the example with this command:

```
kubectl apply -f https://kubernetes.io/examples/controllers/
job.yaml
```

```
job.batch/pi created
```

Check on the status of the Job with kubectl:

```
kubectl describe jobs/pi
```

```

Name:           pi
Namespace:      default
Selector:       controller-uid=c9948307-e56d-4b5d-8302-
ae2d7b7da67c
Labels:         controller-uid=c9948307-e56d-4b5d-8302-
ae2d7b7da67c
                job-name=pi
Annotations:   kubectl.kubernetes.io/last-applied-configuration:
                  {"apiVersion":"batch/
v1","kind":"Job","metadata":{"annotations":
{},"name":"pi","namespace":"default"},"spec":{"backoffLimit":4,"template":...
Parallelism:   1
Completions:   1
Start Time:    Mon, 02 Dec 2019 15:20:11 +0200
Completed At:  Mon, 02 Dec 2019 15:21:16 +0200
Duration:      65s
Pods Statuses: 0 Running / 1 Succeeded / 0 Failed
Pod Template:
  Labels:  controller-uid=c9948307-e56d-4b5d-8302-ae2d7b7da67c
            job-name=pi
  Containers:
    pi:
      Image:      perl
      Port:       <none>
      Host Port: <none>

```

```

Command:
perl
-Mbignum=bpi
-wle
print bpi(2000)
Environment: <none>
Mounts: <none>
Volumes: <none>
Events:
Type Reason Age From Message
---- ---- - - -
Normal SuccessfulCreate 14m job-controller Created pod:
pi-5rwd7

```

To view completed Pods of a Job, use `kubectl get pods`.

To list all the Pods that belong to a Job in a machine readable form, you can use a command like this:

```

pods=$(kubectl get pods --selector=job-name=pi --
output=jsonpath='{.items[*].metadata.name}')
echo $pods

```

`pi-5rwd7`

Here, the selector is the same as the selector for the Job. The `--output=jsonpath` option specifies an expression that just gets the name from each Pod in the returned list.

View the standard output of one of the pods:

```
kubectl logs $pods
```

The output is similar to this:

```

3.141592653589793238462643383279502884197169399375105820974944592
30781640628620899862803482534211706798214808651328230664709384460
95505822317253594081284811174502841027019385211055596446229489549
30381964428810975665933446128475648233786783165271201909145648566
92346034861045432664821339360726024914127372458700660631558817488
15209209628292540917153643678925903600113305305488204665213841469
51941511609433057270365759591953092186117381932611793105118548074
46237996274956735188575272489122793818301194912983367336244065664
30860213949463952247371907021798609437027705392171762931767523846
74818467669405132000568127145263560827785771342757789609173637178
72146844090122495343014654958537105079227968925892354201995611212
90219608640344181598136297747713099605187072113499999983729780499
51059731732816096318595024459455346908302642522308253344685035261
93118817101000313783875288658753320838142061717766914730359825349
04287554687311595628638823537875937519577818577805321712268066130
01927876611195909216420198938095257201065485863278865936153381827
96823030195203530185296899577362259941389124972177528347913151557
48572424541506959508295331168617278558890750983817546374649393192

```

```
55060400927701671139009848824012858361603563707660104710181942955
59619894676783744944825537977472684710404753464620804668425906949
12933136770289891521047521620569660240580381501935112533824300355
87640247496473263914199272604269922796782354781636009341721641219
9245863150302861829745557067498385054945885869269569092721079750
93029553211653449872027559602364806654991198818347977535663698074
26542527862551818417574672890977772793800081647060016145249192173
21721477235014144197356854816136115735255213347574184946843852332
39073941433345477624168625189835694855620992192221842725502542568
87671790494601653466804988627232791786085784383827967976681454100
95388378636095068006422512520511739298489608412848862694560424196
52850222106611863067442786220391949450471237137869609563643719172
874677646575739624138908658326459958133904780275901
```

Writing a Job spec

As with all other Kubernetes config, a Job needs `apiVersion`, `kind`, and `meta` data fields. Its name must be a valid [DNS subdomain name](#).

A Job also needs a [.spec section](#).

Pod Template

The `.spec.template` is the only required field of the `.spec`.

The `.spec.template` is a [pod template](#). It has exactly the same schema as a [Pod](#), except it is nested and does not have an `apiVersion` or `kind`.

In addition to required fields for a Pod, a pod template in a Job must specify appropriate labels (see [pod selector](#)) and an appropriate restart policy.

Only a [RestartPolicy](#) equal to `Never` or `OnFailure` is allowed.

Pod selector

The `.spec.selector` field is optional. In almost all cases you should not specify it. See section [specifying your own pod selector](#).

Parallel execution for Jobs

There are three main types of task suitable to run as a Job:

1. Non-parallel Jobs
 - normally, only one Pod is started, unless the Pod fails.
 - the Job is complete as soon as its Pod terminates successfully.
2. Parallel Jobs with a fixed completion count:
 - specify a non-zero positive value for `.spec.completions`.
 - the Job represents the overall task, and is complete when there is one successful Pod for each value in the range 1 to `.spec.completions`.

- **not implemented yet:** Each Pod is passed a different index in the range 1 to `.spec.completions`.
3. Parallel Jobs with a work queue:
- do not specify `.spec.completions`, default to `.spec.parallelism`.
 - the Pods must coordinate amongst themselves or an external service to determine what each should work on. For example, a Pod might fetch a batch of up to N items from the work queue.
 - each Pod is independently capable of determining whether or not all its peers are done, and thus that the entire Job is done.
 - when any Pod from the Job terminates with success, no new Pods are created.
 - once at least one Pod has terminated with success and all Pods are terminated, then the Job is completed with success.
 - once any Pod has exited with success, no other Pod should still be doing any work for this task or writing any output. They should all be in the process of exiting.

For a non-parallel Job, you can leave both `.spec.completions` and `.spec.parallelism` unset. When both are unset, both are defaulted to 1.

For a fixed completion count Job, you should set `.spec.completions` to the number of completions needed. You can set `.spec.parallelism`, or leave it unset and it will default to 1.

For a work queue Job, you must leave `.spec.completions` unset, and set `.spec.parallelism` to a non-negative integer.

For more information about how to make use of the different types of job, see the [job patterns](#) section.

Controlling parallelism

The requested parallelism (`.spec.parallelism`) can be set to any non-negative value. If it is unspecified, it defaults to 1. If it is specified as 0, then the Job is effectively paused until it is increased.

Actual parallelism (number of pods running at any instant) may be more or less than requested parallelism, for a variety of reasons:

- For fixed completion count Jobs, the actual number of pods running in parallel will not exceed the number of remaining completions. Higher values of `.spec.parallelism` are effectively ignored.
- For work queue Jobs, no new Pods are started after any Pod has succeeded -- remaining Pods are allowed to complete, however.
- If the Job [Controller](#) has not had time to react.
- If the Job controller failed to create Pods for any reason (lack of ResourceQuota, lack of permission, etc.), then there may be fewer pods than requested.
- The Job controller may throttle new Pod creation due to excessive previous pod failures in the same Job.
- When a Pod is gracefully shut down, it takes time to stop.

Handling Pod and container failures

A container in a Pod may fail for a number of reasons, such as because the process in it exited with a non-zero exit code, or the container was killed for exceeding a memory limit, etc. If this happens, and the `.spec.template.spec.restartPolicy = "OnFailure"`, then the Pod stays on the node, but the container is re-run. Therefore, your program needs to handle the case when it is restarted locally, or else specify `.spec.template.spec.restartPolicy = "Never"`. See [pod lifecycle](#) for more information on `restartPolicy`.

An entire Pod can also fail, for a number of reasons, such as when the pod is kicked off the node (node is upgraded, rebooted, deleted, etc.), or if a container of the Pod fails and the `.spec.template.spec.restartPolicy = "Never"`. When a Pod fails, then the Job controller starts a new Pod. This means that your application needs to handle the case when it is restarted in a new pod. In particular, it needs to handle temporary files, locks, incomplete output and the like caused by previous runs.

Note that even if you specify `.spec.parallelism = 1` and `.spec.completions = 1` and `.spec.template.spec.restartPolicy = "Never"`, the same program may sometimes be started twice.

If you do specify `.spec.parallelism` and `.spec.completions` both greater than 1, then there may be multiple pods running at once. Therefore, your pods must also be tolerant of concurrency.

Pod backoff failure policy

There are situations where you want to fail a Job after some amount of retries due to a logical error in configuration etc. To do so, set `.spec.backoffLimit` to specify the number of retries before considering a Job as failed. The back-off limit is set by default to 6. Failed Pods associated with the Job are recreated by the Job controller with an exponential back-off delay (10s, 20s, 40s ...) capped at six minutes. The back-off count is reset when a Job's Pod is deleted or successful without any other Pods for the Job failing around that time.

Note: If your job has `restartPolicy = "OnFailure"`, keep in mind that your container running the Job will be terminated once the job backoff limit has been reached. This can make debugging the Job's executable more difficult. We suggest setting `restartPolicy = "Never"` when debugging the Job or using a logging system to ensure output from failed Jobs is not lost inadvertently.

Job termination and cleanup

When a Job completes, no more Pods are created, but the Pods are not deleted either. Keeping them around allows you to still view the logs of completed pods to check for errors, warnings, or other diagnostic output. The job object also remains after it is completed so that you can view its status. It is up to the user to delete old jobs after noting their status. Delete

the job with `kubectl` (e.g. `kubectl delete jobs/pi` or `kubectl delete -f ./job.yaml`). When you delete the job using `kubectl`, all the pods it created are deleted too.

By default, a Job will run uninterrupted unless a Pod fails (`restartPolicy=Never`) or a Container exits in error (`restartPolicy=OnFailure`), at which point the Job defers to the `.spec.backoffLimit` described above. Once `.spec.backoffLimit` has been reached the Job will be marked as failed and any running Pods will be terminated.

Another way to terminate a Job is by setting an active deadline. Do this by setting the `.spec.activeDeadlineSeconds` field of the Job to a number of seconds. The `activeDeadlineSeconds` applies to the duration of the job, no matter how many Pods are created. Once a Job reaches `activeDeadlineSeconds`, all of its running Pods are terminated and the Job status will become type: Failed with reason: DeadlineExceeded.

Note that a Job's `.spec.activeDeadlineSeconds` takes precedence over its `.spec.backoffLimit`. Therefore, a Job that is retrying one or more failed Pods will not deploy additional Pods once it reaches the time limit specified by `activeDeadlineSeconds`, even if the `backoffLimit` is not yet reached.

Example:

```
apiVersion: batch/v1
kind: Job
metadata:
  name: pi-with-timeout
spec:
  backoffLimit: 5
  activeDeadlineSeconds: 100
  template:
    spec:
      containers:
        - name: pi
          image: perl
          command: ["perl", "-Mbignum=bpi", "-wle", "print
bpi(2000)"]
      restartPolicy: Never
```

Note that both the Job spec and the [Pod template spec](#) within the Job have an `activeDeadlineSeconds` field. Ensure that you set this field at the proper level.

Keep in mind that the `restartPolicy` applies to the Pod, and not to the Job itself: there is no automatic Job restart once the Job status is type: Failed. That is, the Job termination mechanisms activated with `.spec.activeDeadlineSeconds` and `.spec.backoffLimit` result in a permanent Job failure that requires manual intervention to resolve.

Clean up finished jobs automatically

Finished Jobs are usually no longer needed in the system. Keeping them around in the system will put pressure on the API server. If the Jobs are managed directly by a higher level controller, such as [CronJobs](#), the Jobs can be cleaned up by CronJobs based on the specified capacity-based cleanup policy.

TTL mechanism for finished Jobs

FEATURE STATE: Kubernetes v1.12 [alpha]

Another way to clean up finished Jobs (either Complete or Failed) automatically is to use a TTL mechanism provided by a [TTL controller](#) for finished resources, by specifying the `.spec.ttlSecondsAfterFinished` field of the Job.

When the TTL controller cleans up the Job, it will delete the Job cascadingly, i.e. delete its dependent objects, such as Pods, together with the Job. Note that when the Job is deleted, its lifecycle guarantees, such as finalizers, will be honored.

For example:

```
apiVersion: batch/v1
kind: Job
metadata:
  name: pi-with-ttl
spec:
  ttlSecondsAfterFinished: 100
  template:
    spec:
      containers:
        - name: pi
          image: perl
          command: ["perl", "-Mbignum=bpi", "-wle", "print
bpi(2000)"]
      restartPolicy: Never
```

The Job `pi-with-ttl` will be eligible to be automatically deleted, 100 seconds after it finishes.

If the field is set to 0, the Job will be eligible to be automatically deleted immediately after it finishes. If the field is unset, this Job won't be cleaned up by the TTL controller after it finishes.

Note that this TTL mechanism is alpha, with feature gate `TTLAfterFinished`. For more information, see the documentation for [TTL controller](#) for finished resources.

Job patterns

The `Job` object can be used to support reliable parallel execution of Pods. The `Job` object is not designed to support closely-communicating parallel processes, as commonly found in scientific computing. It does support parallel processing of a set of independent but related work items. These might be emails to be sent, frames to be rendered, files to be transcoded, ranges of keys in a NoSQL database to scan, and so on.

In a complex system, there may be multiple different sets of work items. Here we are just considering one set of work items that the user wants to manage together — a batch job.

There are several different patterns for parallel computation, each with strengths and weaknesses. The tradeoffs are:

- One `Job` object for each work item, vs. a single `Job` object for all work items. The latter is better for large numbers of work items. The former creates some overhead for the user and for the system to manage large numbers of `Job` objects.
- Number of pods created equals number of work items, vs. each Pod can process multiple work items. The former typically requires less modification to existing code and containers. The latter is better for large numbers of work items, for similar reasons to the previous bullet.
- Several approaches use a work queue. This requires running a queue service, and modifications to the existing program or container to make it use the work queue. Other approaches are easier to adapt to an existing containerised application.

The tradeoffs are summarized here, with columns 2 to 4 corresponding to the above tradeoffs. The pattern names are also links to examples and more detailed description.

Pattern	Single Job object	Fewer pods than work items?	Use app unmodified?	Works in Kube 1.1?
Job Template Expansion			“	“
Queue with Pod Per Work Item	“		sometimes	“
Queue with Variable Pod Count	“	“		“
Single Job with Static Work Assignment	“		“	

When you specify completions with `.spec.completions`, each Pod created by the `Job` controller has an identical [`spec`](#). This means that all pods for a task will have the same command line and the same image, the same volumes, and (almost) the same environment variables. These patterns are different ways to arrange for pods to work on different things.

This table shows the required settings for `.spec.parallelism` and `.spec.completions` for each of the patterns. Here, W is the number of work items.

Pattern	<code>.spec.completions</code>	<code>.spec.parallelism</code>
Job Template Expansion	1	should be 1
Queue with Pod Per Work Item	W	any
Queue with Variable Pod Count	1	any
Single Job with Static Work Assignment	W	any

Advanced usage

Specifying your own Pod selector

Normally, when you create a `Job` object, you do not specify `.spec.selector`. The system defaulting logic adds this field when the `Job` is created. It picks a selector value that will not overlap with any other jobs.

However, in some cases, you might need to override this automatically set selector. To do this, you can specify the `.spec.selector` of the `Job`.

Be very careful when doing this. If you specify a label selector which is not unique to the pods of that `Job`, and which matches unrelated Pods, then pods of the unrelated job may be deleted, or this `Job` may count other Pods as completing it, or one or both `Jobs` may refuse to create Pods or run to completion. If a non-unique selector is chosen, then other controllers (e.g. `ReplicationController`) and their Pods may behave in unpredictable ways too. Kubernetes will not stop you from making a mistake when specifying `.spec.selector`.

Here is an example of a case when you might want to use this feature.

Say `Job old` is already running. You want existing Pods to keep running, but you want the rest of the Pods it creates to use a different pod template and for the `Job` to have a new name. You cannot update the `Job` because these fields are not updatable. Therefore, you delete `Job old` but leave its pods running, using `kubectl delete jobs/old --cascade=false`. Before deleting it, you make a note of what selector it uses:

```
kubectl get job old -o yaml
```

```
kind: Job
metadata:
  name: old
  ...
spec:
  selector:
    matchLabels:
      controller-uid: a8f3d00d-c6d2-11e5-9f87-42010af00002
  ...
```

Then you create a new Job with name `new` and you explicitly specify the same selector. Since the existing Pods have label `controller-uid=a8f3d00d-c6d2-11e5-9f87-42010af00002`, they are controlled by Job `new` as well.

You need to specify `manualSelector: true` in the new Job since you are not using the selector that the system normally generates for you automatically.

```
kind: Job
metadata:
  name: new
  ...
spec:
  manualSelector: true
  selector:
    matchLabels:
      controller-uid: a8f3d00d-c6d2-11e5-9f87-42010af00002
  ...
```

The new Job itself will have a different uid from `a8f3d00d-c6d2-11e5-9f87-42010af00002`. Setting `manualSelector: true` tells the system to that you know what you are doing and to allow this mismatch.

Alternatives

Bare Pods

When the node that a Pod is running on reboots or fails, the pod is terminated and will not be restarted. However, a Job will create new Pods to replace terminated ones. For this reason, we recommend that you use a Job rather than a bare Pod, even if your application requires only a single Pod.

Replication Controller

Jobs are complementary to [Replication Controllers](#). A Replication Controller manages Pods which are not expected to terminate (e.g. web servers), and a Job manages Pods that are expected to terminate (e.g. batch tasks).

As discussed in [Pod Lifecycle](#), Job is only appropriate for pods with Restart Policy equal to OnFailure or Never. (Note: If `RestartPolicy` is not set, the default value is Always.)

Single Job starts controller Pod

Another pattern is for a single Job to create a Pod which then creates other Pods, acting as a sort of custom controller for those Pods. This allows the most flexibility, but may be somewhat complicated to get started with and offers less integration with Kubernetes.

One example of this pattern would be a Job which starts a Pod which runs a script that in turn starts a Spark master controller (see [spark example](#)), runs a spark driver, and then cleans up.

An advantage of this approach is that the overall process gets the completion guarantee of a Job object, but maintains complete control over what Pods are created and how work is assigned to them.

Cron Jobs

You can use a [CronJob](#) to create a Job that will run at specified times/dates, similar to the Unix tool `cron`.

Feedback

Was this page helpful?

Yes No

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

Last modified August 10, 2020 at 9:56 AM PST: [Improve order of workload resources \(7f1a2cace\)](#)

[Edit this page](#) [Create child page](#) [Create an issue](#)

- [Running an example Job](#)
- [Writing a Job spec](#)
 - [Pod Template](#)
 - [Pod selector](#)
 - [Parallel execution for Jobs](#)
- [Handling Pod and container failures](#)
 - [Pod backoff failure policy](#)
- [Job termination and cleanup](#)
- [Clean up finished jobs automatically](#)
 - [TTL mechanism for finished Jobs](#)
- [Job patterns](#)
- [Advanced usage](#)
 - [Specifying your own Pod selector](#)
- [Alternatives](#)
 - [Bare Pods](#)
 - [Replication Controller](#)
 - [Single Job starts controller Pod](#)
- [Cron Jobs](#)

Garbage Collection

The role of the Kubernetes garbage collector is to delete certain objects that once had an owner, but no longer have an owner.

Owners and dependents

Some Kubernetes objects are owners of other objects. For example, a `ReplicaSet` is the owner of a set of `Pods`. The owned objects are called dependents of the owner object. Every dependent object has a `metadata.ownerReferences` field that points to the owning object.

Sometimes, Kubernetes sets the value of `ownerReference` automatically. For example, when you create a `ReplicaSet`, Kubernetes automatically sets the `ownerReference` field of each `Pod` in the `ReplicaSet`. In 1.8, Kubernetes automatically sets the value of `ownerReference` for objects created or adopted by `ReplicationController`, `ReplicaSet`, `StatefulSet`, `DaemonSet`, `Deployment`, `Job` and `CronJob`.

You can also specify relationships between owners and dependents by manually setting the `ownerReference` field.

Here's a configuration file for a `ReplicaSet` that has three `Pods`:

[controllers/replicaset.yaml](#)



```
apiVersion: apps/v1
kind: ReplicaSet
metadata:
  name: my-repset
spec:
  replicas: 3
  selector:
    matchLabels:
      pod-is-for: garbage-collection-example
  template:
    metadata:
      labels:
        pod-is-for: garbage-collection-example
    spec:
      containers:
        - name: nginx
          image: nginx
```

If you create the `ReplicaSet` and then view the `Pod` metadata, you can see `OwnerReferences` field:

```
kubectl apply -f https://k8s.io/examples/controllers/
replicaset.yaml
kubectl get pods --output=yaml
```

The output shows that the `Pod` owner is a `ReplicaSet` named `my-repset`:

```
apiVersion: v1
kind: Pod
metadata:
```

```
  ...
  ownerReferences:
  - apiVersion: apps/v1
    controller: true
    blockOwnerDeletion: true
    kind: ReplicaSet
    name: my-repset
    uid: d9607e19-f88f-11e6-a518-42010a800195
  ...

```

Note:

Cross-namespace owner references are disallowed by design.

*Namespaced dependents can specify cluster-scoped or namespaced owners. A namespaced owner **must** exist in the same namespace as the dependent. If it does not, the owner reference is treated as absent, and the dependent is subject to deletion once all owners are verified absent.*

Cluster-scoped dependents can only specify cluster-scoped owners. In v1.20+, if a cluster-scoped dependent specifies a namespaced kind as an owner, it is treated as having an unresolvable owner reference, and is not able to be garbage collected.

In v1.20+, if the garbage collector detects an invalid cross-namespace ownerReference, or a cluster-scoped dependent with an ownerReference referencing a namespaced kind, a warning Event with a reason of OwnerRefInvalidNamespace and an involvedObject of the invalid dependent is reported. You can check for that kind of Event by running kubectl get events -A --field-selector=reason=OwnerRefInvalidNamespace.

Controlling how the garbage collector deletes dependents

When you delete an object, you can specify whether the object's dependents are also deleted automatically. Deleting dependents automatically is called cascading deletion. There are two modes of cascading deletion: background and foreground.

If you delete an object without deleting its dependents automatically, the dependents are said to be orphaned.

Foreground cascading deletion

In foreground cascading deletion, the root object first enters a "deletion in progress" state. In the "deletion in progress" state, the following things are true:

- The object is still visible via the REST API
- The object's `deletionTimestamp` is set
- The object's `metadata.finalizers` contains the value "foregroundDeletion".

Once the "deletion in progress" state is set, the garbage collector deletes the object's dependents. Once the garbage collector has deleted all "blocking" dependents (objects with `ownerReference.blockOwnerDeletion=true`), it deletes the owner object.

Note that in the "foregroundDeletion", only dependents with `ownerReference.blockOwnerDeletion=true` block the deletion of the owner object.

Kubernetes version 1.7 added an [admission controller](#) that controls user access to set `blockOwnerDeletion` to true based on delete permissions on the owner object, so that unauthorized dependents cannot delay deletion of an owner object.

If an object's `ownerReferences` field is set by a controller (such as Deployment or ReplicaSet), `blockOwnerDeletion` is set automatically and you do not need to manually modify this field.

Background cascading deletion

In background cascading deletion, Kubernetes deletes the owner object immediately and the garbage collector then deletes the dependents in the background.

Setting the cascading deletion policy

To control the cascading deletion policy, set the `propagationPolicy` field on the `deleteOptions` argument when deleting an Object. Possible values include "Orphan", "Foreground", or "Background".

Here's an example that deletes dependents in background:

```
kubectl proxy --port=8080
curl -X DELETE localhost:8080/apis/apps/v1/namespaces/default/
replicaset/my-repset \
-d '{"kind":"DeleteOptions","apiVersion":"v1","propagationPolicy":"Background"}' \
-H "Content-Type: application/json"
```

Here's an example that deletes dependents in foreground:

```
kubectl proxy --port=8080
curl -X DELETE localhost:8080/apis/apps/v1/namespaces/default/
```

```
replicasets/my-repset \
  -d '{"kind": "DeleteOptions", "apiVersion": "v1", "propagationPolicy": "Foreground"}' \
  -H "Content-Type: application/json"
```

Here's an example that orphans dependents:

```
kubectl proxy --port=8080
curl -X DELETE localhost:8080/apis/apps/v1/namespaces/default/
replicasets/my-repset \
  -d '{"kind": "DeleteOptions", "apiVersion": "v1", "propagationPolicy": "Orphan"}' \
  -H "Content-Type: application/json"
```

kubectl also supports cascading deletion. To delete dependents automatically using *kubectl*, set `--cascade` to true. To orphan dependents, set `--cascade` to false. The default value for `--cascade` is true.

Here's an example that orphans the dependents of a ReplicaSet:

```
kubectl delete replicaset my-repset --cascade=false
```

Additional note on Deployments

Prior to 1.7, When using cascading deletes with Deployments you must use `propagationPolicy: Foreground` to delete not only the ReplicaSets created, but also their Pods. If this type of propagationPolicy is not used, only the ReplicaSets will be deleted, and the Pods will be orphaned. See [kubeadm/#149](#) for more information.

Known issues

Tracked at [#26120](#)

What's next

[Design Doc 1](#)

[Design Doc 2](#)

Feedback

Was this page helpful?

Yes No

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

Last modified November 17, 2020 at 5:38 PM PST: [Update GC cross-namespace note \(8d96fcb42\)](#)
[Edit this page](#) [Create child page](#) [Create an issue](#)

- [Owners and dependents](#)
- [Controlling how the garbage collector deletes dependents](#)
 - [Foreground cascading deletion](#)
 - [Background cascading deletion](#)
 - [Setting the cascading deletion policy](#)
 - [Additional note on Deployments](#)
- [Known issues](#)
- [What's next](#)

TTL Controller for Finished Resources

FEATURE STATE: Kubernetes v1.12 [alpha]

The TTL controller provides a TTL (time to live) mechanism to limit the lifetime of resource objects that have finished execution. TTL controller only handles [Jobs](#) for now, and may be expanded to handle other resources that will finish execution, such as Pods and custom resources.

Alpha Disclaimer: this feature is currently alpha, and can be enabled with both kube-apiserver and kube-controller-manager [feature gate](#) TTLAfterFinished.

TTL Controller

The TTL controller only supports Jobs for now. A cluster operator can use this feature to clean up finished Jobs (either Complete or Failed) automatically by specifying the `.spec.ttlSecondsAfterFinished` field of a Job, as in this [example](#). The TTL controller will assume that a resource is eligible to be cleaned up TTL seconds after the resource has finished, in other words, when the TTL has expired. When the TTL controller cleans up a resource, it will delete it cascadingly, that is to say it will delete its dependent objects together with it. Note that when the resource is deleted, its lifecycle guarantees, such as finalizers, will be honored.

The TTL seconds can be set at any time. Here are some examples for setting the `.spec.ttlSecondsAfterFinished` field of a Job:

- Specify this field in the resource manifest, so that a Job can be cleaned up automatically some time after it finishes.
- Set this field of existing, already finished resources, to adopt this new feature.
- Use a [mutating admission webhook](#) to set this field dynamically at resource creation time. Cluster administrators can use this to enforce a TTL policy for finished resources.

- Use a [mutating admission webhook](#) to set this field dynamically after the resource has finished, and choose different TTL values based on resource status, labels, etc.

Caveat

Updating TTL Seconds

Note that the TTL period, e.g. `.spec.ttlSecondsAfterFinished` field of Jobs, can be modified after the resource is created or has finished. However, once the Job becomes eligible to be deleted (when the TTL has expired), the system won't guarantee that the Jobs will be kept, even if an update to extend the TTL returns a successful API response.

Time Skew

Because TTL controller uses timestamps stored in the Kubernetes resources to determine whether the TTL has expired or not, this feature is sensitive to time skew in the cluster, which may cause TTL controller to clean up resource objects at the wrong time.

In Kubernetes, it's required to run NTP on all nodes (see [#6159](#)) to avoid time skew. Clocks aren't always correct, but the difference should be very small. Please be aware of this risk when setting a non-zero TTL.

What's next

- [Clean up Jobs automatically](#)
- [Design doc](#)

Feedback

Was this page helpful?

Yes No

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

Last modified July 17, 2020 at 3:46 PM PST: [Replace reference to redirect entries \(1\) \(0bdcd44e6\)](#)

[Edit this page](#) [Create child page](#) [Create an issue](#)

- [TTL Controller](#)
- [Caveat](#)
 - [Updating TTL Seconds](#)
 - [Time Skew](#)
- [What's next](#)

CronJob

FEATURE STATE: Kubernetes v1.8 [beta]

A CronJob creates [Jobs](#) on a repeating schedule.

One CronJob object is like one line of a crontab (cron table) file. It runs a job periodically on a given schedule, written in [Cron](#) format.

Caution:

All **CronJob** schedule: times are based on the timezone of the [kube-controller-manager](#).

If your control plane runs the kube-controller-manager in Pods or bare containers, the timezone set for the kube-controller-manager container determines the timezone that the cron job controller uses.

When creating the manifest for a CronJob resource, make sure the name you provide is a valid [DNS subdomain name](#). The name must be no longer than 52 characters. This is because the CronJob controller will automatically append 11 characters to the job name provided and there is a constraint that the maximum length of a Job name is no more than 63 characters.

CronJob

CronJobs are useful for creating periodic and recurring tasks, like running backups or sending emails. CronJobs can also schedule individual tasks for a specific time, such as scheduling a Job for when your cluster is likely to be idle.

Example

This example CronJob manifest prints the current time and a hello message every minute:

[application/job/cronjob.yaml](#)



```
apiVersion: batch/v1beta1
kind: CronJob
metadata:
  name: hello
spec:
  schedule: "*/1 * * * *"
  jobTemplate:
    spec:
      template:
```

```

spec:
  containers:
    - name: hello
      image: busybox
      imagePullPolicy: IfNotPresent
      args:
        - /bin/sh
        - -c
        - date; echo Hello from the Kubernetes cluster
  restartPolicy: OnFailure

```

([Running Automated Tasks with a CronJob](#) takes you through this example in more detail).

CronJob limitations

A cron job creates a job object about once per execution time of its schedule. We say "about" because there are certain circumstances where two jobs might be created, or no job might be created. We attempt to make these rare, but do not completely prevent them. Therefore, jobs should be idempotent.

If `startingDeadlineSeconds` is set to a large value or left unset (the default) and if `concurrencyPolicy` is set to `Allow`, the jobs will always run at least once.

For every CronJob, the CronJob [Controller](#) checks how many schedules it missed in the duration from its last scheduled time until now. If there are more than 100 missed schedules, then it does not start the job and logs the error

Cannot determine if job needs to be started. Too many missed start time (> 100). Set or decrease .spec.startingDeadlineSeconds or check clock skew.

It is important to note that if the `startingDeadlineSeconds` field is set (not nil), the controller counts how many missed jobs occurred from the value of `startingDeadlineSeconds` until now rather than from the last scheduled time until now. For example, if `startingDeadlineSeconds` is 200, the controller counts how many missed jobs occurred in the last 200 seconds.

A CronJob is counted as missed if it has failed to be created at its scheduled time. For example, If `concurrencyPolicy` is set to `Forbid` and a CronJob was attempted to be scheduled when there was a previous schedule still running, then it would count as missed.

For example, suppose a CronJob is set to schedule a new Job every one minute beginning at 08:30:00, and its `startingDeadlineSeconds` field is not set. If the CronJob controller happens to be down from 08:29:00 to 10:21:00, the job will not start as the number of missed jobs which missed their schedule is greater than 100.

To illustrate this concept further, suppose a CronJob is set to schedule a new Job every one minute beginning at 08:30:00, and its `startingDeadlineSeconds` is set to 200 seconds. If the CronJob controller happens to be down for the same period as the previous example (08:29:00 to 10:21:00,) the Job will still start at 10:22:00. This happens as the controller now checks how many missed schedules happened in the last 200 seconds (ie, 3 missed schedules), rather than from the last scheduled time until now.

The CronJob is only responsible for creating Jobs that match its schedule, and the Job in turn is responsible for the management of the Pods it represents.

New controller

There's an alternative implementation of the CronJob controller, available as an alpha feature since Kubernetes 1.20. To select version 2 of the CronJob controller, pass the following [feature gate](#) flag to the [kube-controller-manager](#).

```
--feature-gates="CronJobControllerV2=true"
```

What's next

[Cron expression format](#) documents the format of CronJob schedule fields.

For instructions on creating and working with cron jobs, and for an example of CronJob manifest, see [Running automated tasks with cron jobs](#).

Feedback

Was this page helpful?

Yes No

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

Last modified November 04, 2020 at 5:54 PM PST: [Add information how to enable cronjob controller v2 \(4f0068f33\)](#)

[Edit this page](#) [Create child page](#) [Create an issue](#)

- [CronJob](#)
 - [Example](#)
- [CronJob limitations](#)
- [New controller](#)
- [What's next](#)

ReplicationController

Note: A [Deployment](#) that configures a [ReplicaSet](#) is now the recommended way to set up replication.

A ReplicationController ensures that a specified number of pod replicas are running at any one time. In other words, a ReplicationController makes sure that a pod or a homogeneous set of pods is always up and available.

How a ReplicationController Works

If there are too many pods, the ReplicationController terminates the extra pods. If there are too few, the ReplicationController starts more pods. Unlike manually created pods, the pods maintained by a ReplicationController are automatically replaced if they fail, are deleted, or are terminated. For example, your pods are re-created on a node after disruptive maintenance such as a kernel upgrade. For this reason, you should use a ReplicationController even if your application requires only a single pod. A ReplicationController is similar to a process supervisor; but instead of supervising individual processes on a single node, the ReplicationController supervises multiple pods across multiple nodes.

ReplicationController is often abbreviated to "rc" in discussion, and as a shortcut in kubectl commands.

A simple case is to create one ReplicationController object to reliably run one instance of a Pod indefinitely. A more complex use case is to run several identical replicas of a replicated service, such as web servers.

Running an example ReplicationController

This example ReplicationController config runs three copies of the nginx web server.

[controllers/replication.yaml](#)



```
apiVersion: v1
kind: ReplicationController
metadata:
  name: nginx
spec:
  replicas: 3
  selector:
    app: nginx
  template:
    metadata:
      name: nginx
    labels:
```

```

  app: nginx
spec:
  containers:
    - name: nginx
      image: nginx
      ports:
        - containerPort: 80

```

Run the example job by downloading the example file and then running this command:

```
kubectl apply -f https://k8s.io/examples/controllers/
replication.yaml
```

```
replicationcontroller/nginx created
```

Check on the status of the ReplicationController using this command:

```
kubectl describe replicationcontrollers/nginx
```

```

Name:          nginx
Namespace:     default
Selector:      app=nginx
Labels:        app=nginx
Annotations:   <none>
Replicas:      3 current / 3 desired
Pods Status:   0 Running / 3 Waiting / 0 Succeeded / 0 Failed
Pod Template:
  Labels:        app=nginx
  Containers:
    nginx:
      Image:       nginx
      Port:        80/TCP
      Environment: <none>
      Mounts:      <none>
      Volumes:     <none>
  Events:
    FirstSeen     LastSeen      Count
    From          SubobjectPath  Type
    Reason        Message
    ----          -----        -----
    20s           20s          1   Normal       {replication-
controller }                           SuccessfulCreate
    Created pod: nginx-qrm3m
    20s           20s          1   Normal       {replication-
controller }                           SuccessfulCreate
    Created pod: nginx-3ntk0
    20s           20s          1   Normal       {replication-
controller }                           SuccessfulCreate
    Created pod: nginx-4ok8v

```

Here, three pods are created, but none is running yet, perhaps because the image is being pulled. A little later, the same command may show:

```
Pods Status:    3 Running / 0 Waiting / 0 Succeeded / 0 Failed
```

To list all the pods that belong to the ReplicationController in a machine readable form, you can use a command like this:

```
pods=$(kubectl get pods --selector=app=nginx --output=jsonpath='{.items..metadata.name}')
echo $pods
```

```
nginx-3ntk0 nginx-4ok8v nginx-qrm3m
```

Here, the selector is the same as the selector for the ReplicationController (seen in the kubectl describe output), and in a different form in replication.yaml. The --output=jsonpath option specifies an expression that just gets the name from each pod in the returned list.

Writing a ReplicationController Spec

As with all other Kubernetes config, a ReplicationController needs `apiVersion`, `kind`, and `metadata` fields. The name of a ReplicationController object must be a valid [DNS subdomain name](#). For general information about working with config files, see [object management](#).

A ReplicationController also needs a [.spec section](#).

Pod Template

The `.spec.template` is the only required field of the `.spec`.

The `.spec.template` is a [pod template](#). It has exactly the same schema as a [Pod](#), except it is nested and does not have an `apiVersion` or `kind`.

In addition to required fields for a Pod, a pod template in a ReplicationController must specify appropriate labels and an appropriate restart policy. For labels, make sure not to overlap with other controllers. See [pod selector](#).

Only a `.spec.template.spec.restartPolicy` equal to `Always` is allowed, which is the default if not specified.

For local container restarts, ReplicationControllers delegate to an agent on the node, for example the [Kubelet](#) or Docker.

Labels on the ReplicationController

The ReplicationController can itself have labels (`.metadata.labels`). Typically, you would set these the same as the `.spec.template.metadata.labels`; if `.metadata.labels` is not specified then it defaults to `.spec.templa`

`te.metadata.labels`. However, they are allowed to be different, and the `.me`
`tadata.labels` do not affect the behavior of the ReplicationController.

Pod Selector

The `.spec.selector` field is a [label selector](#). A ReplicationController manages all the pods with labels that match the selector. It does not distinguish between pods that it created or deleted and pods that another person or process created or deleted. This allows the ReplicationController to be replaced without affecting the running pods.

If specified, the `.spec.template.metadata.labels` must be equal to the `.sp`
`ec.selector`, or it will be rejected by the API. If `.spec.selector` is
unspecified, it will be defaulted to `.spec.template.metadata.labels`.

Also you should not normally create any pods whose labels match this selector, either directly, with another ReplicationController, or with another controller such as Job. If you do so, the ReplicationController thinks that it created the other pods. Kubernetes does not stop you from doing this.

If you do end up with multiple controllers that have overlapping selectors, you will have to manage the deletion yourself (see [below](#)).

Multiple Replicas

You can specify how many pods should run concurrently by setting `.spec.replicas` to the number of pods you would like to have running concurrently. The number running at any time may be higher or lower, such as if the replicas were just increased or decreased, or if a pod is gracefully shutdown, and a replacement starts early.

If you do not specify `.spec.replicas`, then it defaults to 1.

Working with ReplicationControllers

Deleting a ReplicationController and its Pods

To delete a ReplicationController and all its pods, use [`kubectl delete`](#).
Kubectl will scale the ReplicationController to zero and wait for it to delete each pod before deleting the ReplicationController itself. If this kubectl command is interrupted, it can be restarted.

When using the REST API or go client library, you need to do the steps explicitly (scale replicas to 0, wait for pod deletions, then delete the ReplicationController).

Deleting just a ReplicationController

You can delete a ReplicationController without affecting any of its pods.

Using kubectl, specify the `--cascade=false` option to [`kubectl delete`](#).

When using the REST API or go client library, simply delete the ReplicationController object.

Once the original is deleted, you can create a new ReplicationController to replace it. As long as the old and new .spec.selector are the same, then the new one will adopt the old pods. However, it will not make any effort to make existing pods match a new, different pod template. To update pods to a new spec in a controlled way, use a [rolling update](#).

Isolating pods from a ReplicationController

Pods may be removed from a ReplicationController's target set by changing their labels. This technique may be used to remove pods from service for debugging, data recovery, etc. Pods that are removed in this way will be replaced automatically (assuming that the number of replicas is not also changed).

Common usage patterns

Rescheduling

As mentioned above, whether you have 1 pod you want to keep running, or 1000, a ReplicationController will ensure that the specified number of pods exists, even in the event of node failure or pod termination (for example, due to an action by another control agent).

Scaling

The ReplicationController makes it easy to scale the number of replicas up or down, either manually or by an auto-scaling control agent, by simply updating the replicas field.

Rolling updates

The ReplicationController is designed to facilitate rolling updates to a service by replacing pods one-by-one.

As explained in [#1353](#), the recommended approach is to create a new ReplicationController with 1 replica, scale the new (+1) and old (-1) controllers one by one, and then delete the old controller after it reaches 0 replicas. This predictably updates the set of pods regardless of unexpected failures.

Ideally, the rolling update controller would take application readiness into account, and would ensure that a sufficient number of pods were productively serving at any given time.

The two ReplicationControllers would need to create pods with at least one differentiating label, such as the image tag of the primary container of the pod, since it is typically image updates that motivate rolling updates.

Multiple release tracks

In addition to running multiple releases of an application while a rolling update is in progress, it's common to run multiple releases for an extended period of time, or even continuously, using multiple release tracks. The tracks would be differentiated by labels.

For instance, a service might target all pods with tier in (frontend), environment in (prod). Now say you have 10 replicated pods that make up this tier. But you want to be able to 'canary' a new version of this component. You could set up a ReplicationController with replicas set to 9 for the bulk of the replicas, with labels tier=frontend, environment=prod, track=stable, and another ReplicationController with replicas set to 1 for the canary, with labels tier=frontend, environment=prod, track=canary. Now the service is covering both the canary and non-canary pods. But you can mess with the ReplicationControllers separately to test things out, monitor the results, etc.

Using ReplicationControllers with Services

Multiple ReplicationControllers can sit behind a single service, so that, for example, some traffic goes to the old version, and some goes to the new version.

A ReplicationController will never terminate on its own, but it isn't expected to be as long-lived as services. Services may be composed of pods controlled by multiple ReplicationControllers, and it is expected that many ReplicationControllers may be created and destroyed over the lifetime of a service (for instance, to perform an update of pods that run the service). Both services themselves and their clients should remain oblivious to the ReplicationControllers that maintain the pods of the services.

Writing programs for Replication

Pods created by a ReplicationController are intended to be fungible and semantically identical, though their configurations may become heterogeneous over time. This is an obvious fit for replicated stateless servers, but ReplicationControllers can also be used to maintain availability of master-elected, sharded, and worker-pool applications. Such applications should use dynamic work assignment mechanisms, such as the [RabbitMQ work queues](#), as opposed to static/one-time customization of the configuration of each pod, which is considered an anti-pattern. Any pod customization performed, such as vertical auto-sizing of resources (for example, cpu or memory), should be performed by another online controller process, not unlike the ReplicationController itself.

Responsibilities of the ReplicationController

The ReplicationController simply ensures that the desired number of pods matches its label selector and are operational. Currently, only terminated

pods are excluded from its count. In the future, [readiness](#) and other information available from the system may be taken into account, we may add more controls over the replacement policy, and we plan to emit events that could be used by external clients to implement arbitrarily sophisticated replacement and/or scale-down policies.

The ReplicationController is forever constrained to this narrow responsibility. It itself will not perform readiness nor liveness probes. Rather than performing auto-scaling, it is intended to be controlled by an external auto-scaler (as discussed in [#492](#)), which would change its `replicas` field. We will not add scheduling policies (for example, [spreading](#)) to the ReplicationController. Nor should it verify that the pods controlled match the currently specified template, as that would obstruct auto-sizing and other automated processes. Similarly, completion deadlines, ordering dependencies, configuration expansion, and other features belong elsewhere. We even plan to factor out the mechanism for bulk pod creation ([#170](#)).

The ReplicationController is intended to be a composable building-block primitive. We expect higher-level APIs and/or tools to be built on top of it and other complementary primitives for user convenience in the future. The "macro" operations currently supported by kubectl (`run`, `scale`) are proof-of-concept examples of this. For instance, we could imagine something like [Asgard](#) managing ReplicationControllers, auto-scalers, services, scheduling policies, canaries, etc.

API Object

Replication controller is a top-level resource in the Kubernetes REST API. More details about the API object can be found at: [ReplicationController API object](#).

Alternatives to ReplicationController

ReplicaSet

[ReplicaSet](#) is the next-generation ReplicationController that supports the new [set-based label selector](#). It's mainly used by [Deployment](#) as a mechanism to orchestrate pod creation, deletion and updates. Note that we recommend using Deployments instead of directly using Replica Sets, unless you require custom update orchestration or don't require updates at all.

Deployment (Recommended)

[Deployment](#) is a higher-level API object that updates its underlying Replica Sets and their Pods. Deployments are recommended if you want this rolling update functionality because, they are declarative, server-side, and have additional features.

Bare Pods

Unlike in the case where a user directly created pods, a ReplicationController replaces pods that are deleted or terminated for any reason, such as in the case of node failure or disruptive node maintenance, such as a kernel upgrade. For this reason, we recommend that you use a ReplicationController even if your application requires only a single pod. Think of it similarly to a process supervisor, only it supervises multiple pods across multiple nodes instead of individual processes on a single node. A ReplicationController delegates local container restarts to some agent on the node (for example, Kubelet or Docker).

Job

Use a [Job](#) instead of a ReplicationController for pods that are expected to terminate on their own (that is, batch jobs).

DaemonSet

Use a [DaemonSet](#) instead of a ReplicationController for pods that provide a machine-level function, such as machine monitoring or machine logging. These pods have a lifetime that is tied to a machine lifetime: the pod needs to be running on the machine before other pods start, and are safe to terminate when the machine is otherwise ready to be rebooted/shutdown.

For more information

Read [Run Stateless Application Deployment](#).

Feedback

Was this page helpful?

Yes *No*

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

Last modified August 05, 2020 at 3:17 AM PST: [Replace special quote characters with normal ones. \(c6a96128c\)](#)

[Edit this page](#) [Create child page](#) [Create an issue](#)

- [How a ReplicationController Works](#)
- [Running an example ReplicationController](#)
- [Writing a ReplicationController Spec](#)
 - [Pod Template](#)
 - [Labels on the ReplicationController](#)
 - [Pod Selector](#)
 - [Multiple Replicas](#)

- [Working with ReplicationControllers](#)
 - [Deleting a ReplicationController and its Pods](#)
 - [Deleting just a ReplicationController](#)
 - [Isolating pods from a ReplicationController](#)
- [Common usage patterns](#)
 - [Rescheduling](#)
 - [Scaling](#)
 - [Rolling updates](#)
 - [Multiple release tracks](#)
 - [Using ReplicationControllers with Services](#)
- [Writing programs for Replication](#)
- [Responsibilities of the ReplicationController](#)
- [API Object](#)
- [Alternatives to ReplicationController](#)
 - [ReplicaSet](#)
 - [Deployment \(Recommended\)](#)
 - [Bare Pods](#)
 - [Job](#)
 - [DaemonSet](#)
- [For more information](#)

Services, Load Balancing, and Networking

Concepts and resources behind networking in Kubernetes.

Kubernetes networking addresses four concerns:

- *Containers within a Pod use networking to communicate via loopback.*
 - *Cluster networking provides communication between different Pods.*
 - *The Service resource lets you expose an application running in Pods to be reachable from outside your cluster.*
 - *You can also use Services to publish services only for consumption inside your cluster.*
-

[Service](#)

[Service Topology](#)

[DNS for Services and Pods](#)

[Connecting Applications with Services](#)

[EndpointSlices](#)

[Ingress](#)

[Ingress Controllers](#)

Network Policies

Adding entries to Pod /etc/hosts with HostAliases

IPv4/IPv6 dual-stack

Service

An abstract way to expose an application running on a set of [Pods](#) as a network service.

With Kubernetes you don't need to modify your application to use an unfamiliar service discovery mechanism. Kubernetes gives Pods their own IP addresses and a single DNS name for a set of Pods, and can load-balance across them.

Motivation

Kubernetes [Pods](#) are created and destroyed to match the state of your cluster. Pods are nonpermanent resources. If you use a [Deployment](#) to run your app, it can create and destroy Pods dynamically.

Each Pod gets its own IP address, however in a Deployment, the set of Pods running in one moment in time could be different from the set of Pods running that application a moment later.

This leads to a problem: if some set of Pods (call them "backends") provides functionality to other Pods (call them "frontends") inside your cluster, how do the frontends find out and keep track of which IP address to connect to, so that the frontend can use the backend part of the workload?

Enter Services.

Service resources

In Kubernetes, a Service is an abstraction which defines a logical set of Pods and a policy by which to access them (sometimes this pattern is called a micro-service). The set of Pods targeted by a Service is usually determined by a [selector](#). To learn about other ways to define Service endpoints, see [Services without selectors](#).

For example, consider a stateless image-processing backend which is running with 3 replicas. Those replicas are fungible—frontends do not care

which backend they use. While the actual Pods that compose the backend set may change, the frontend clients should not need to be aware of that, nor should they need to keep track of the set of backends themselves.

The Service abstraction enables this decoupling.

Cloud-native service discovery

If you're able to use Kubernetes APIs for service discovery in your application, you can query the [API server](#) for Endpoints, that get updated whenever the set of Pods in a Service changes.

For non-native applications, Kubernetes offers ways to place a network port or load balancer in between your application and the backend Pods.

Defining a Service

A Service in Kubernetes is a REST object, similar to a Pod. Like all of the REST objects, you can POST a Service definition to the API server to create a new instance. The name of a Service object must be a valid [DNS label name](#).

For example, suppose you have a set of Pods that each listen on TCP port 9376 and carry a label app=MyApp:

```
apiVersion: v1
kind: Service
metadata:
  name: my-service
spec:
  selector:
    app: MyApp
  ports:
    - protocol: TCP
      port: 80
      targetPort: 9376
```

This specification creates a new Service object named "my-service", which targets TCP port 9376 on any Pod with the app=MyApp label.

Kubernetes assigns this Service an IP address (sometimes called the "cluster IP"), which is used by the Service proxies (see [Virtual IPs and service proxies](#) below).

The controller for the Service selector continuously scans for Pods that match its selector, and then POSTs any updates to an Endpoint object also named "my-service".

Note: A Service can map any incoming port to a targetPort. By default and for convenience, the targetPort is set to the same value as the port field.

Port definitions in Pods have names, and you can reference these names in the targetPort attribute of a Service. This works even if there is a mixture of Pods in the Service using a single configured name, with the same network protocol available via different port numbers. This offers a lot of flexibility for deploying and evolving your Services. For example, you can change the port numbers that Pods expose in the next version of your backend software, without breaking clients.

The default protocol for Services is TCP; you can also use any other [supported protocol](#).

As many Services need to expose more than one port, Kubernetes supports multiple port definitions on a Service object. Each port definition can have the same protocol, or a different one.

Services without selectors

Services most commonly abstract access to Kubernetes Pods, but they can also abstract other kinds of backends. For example:

- You want to have an external database cluster in production, but in your test environment you use your own databases.
- You want to point your Service to a Service in a different [Namespace](#) or on another cluster.
- You are migrating a workload to Kubernetes. While evaluating the approach, you run only a proportion of your backends in Kubernetes.

In any of these scenarios you can define a Service without a Pod selector. For example:

```
apiVersion: v1
kind: Service
metadata:
  name: my-service
spec:
  ports:
    - protocol: TCP
```

```
port: 80
targetPort: 9376
```

Because this Service has no selector, the corresponding Endpoint object is not created automatically. You can manually map the Service to the network address and port where it's running, by adding an Endpoint object manually:

```
apiVersion: v1
kind: Endpoints
metadata:
  name: my-service
subsets:
- addresses:
  - ip: 192.0.2.42
  ports:
  - port: 9376
```

The name of the Endpoints object must be a valid [DNS subdomain name](#).

Note:

The endpoint IPs must not be: loopback (127.0.0.0/8 for IPv4, ::1/128 for IPv6), or link-local (169.254.0.0/16 and 224.0.0.0/24 for IPv4, fe80::/64 for IPv6).

Endpoint IP addresses cannot be the cluster IPs of other Kubernetes Services, because [kube-proxy](#) doesn't support virtual IPs as a destination.

Accessing a Service without a selector works the same as if it had a selector. In the example above, traffic is routed to the single endpoint defined in the YAML: 192.0.2.42:9376 (TCP).

An ExternalName Service is a special case of Service that does not have selectors and uses DNS names instead. For more information, see the [ExternalName](#) section later in this document.

EndpointSlices

FEATURE STATE: *Kubernetes v1.17 [beta]*

EndpointSlices are an API resource that can provide a more scalable alternative to Endpoints. Although conceptually quite similar to Endpoints, EndpointSlices allow for distributing network endpoints across multiple resources. By default, an EndpointSlice is considered "full" once it reaches

100 endpoints, at which point additional EndpointSlices will be created to store any additional endpoints.

EndpointSlices provide additional attributes and functionality which is described in detail in [EndpointSlices](#).

Application protocol

FEATURE STATE: Kubernetes v1.20 [stable]

The appProtocol field provides a way to specify an application protocol for each Service port. The value of this field is mirrored by the corresponding Endpoints and EndpointSlice objects.

This field follows standard Kubernetes label syntax. Values should either be [IANA standard service names](#) or domain prefixed names such as mycompany.com/my-custom-protocol.

Virtual IPs and service proxies

Every node in a Kubernetes cluster runs a kube-proxy. kube-proxy is responsible for implementing a form of virtual IP for Services of type other than [ExternalName](#).

Why not use round-robin DNS?

A question that pops up every now and then is why Kubernetes relies on proxying to forward inbound traffic to backends. What about other approaches? For example, would it be possible to configure DNS records that have multiple A values (or AAAA for IPv6), and rely on round-robin name resolution?

There are a few reasons for using proxying for Services:

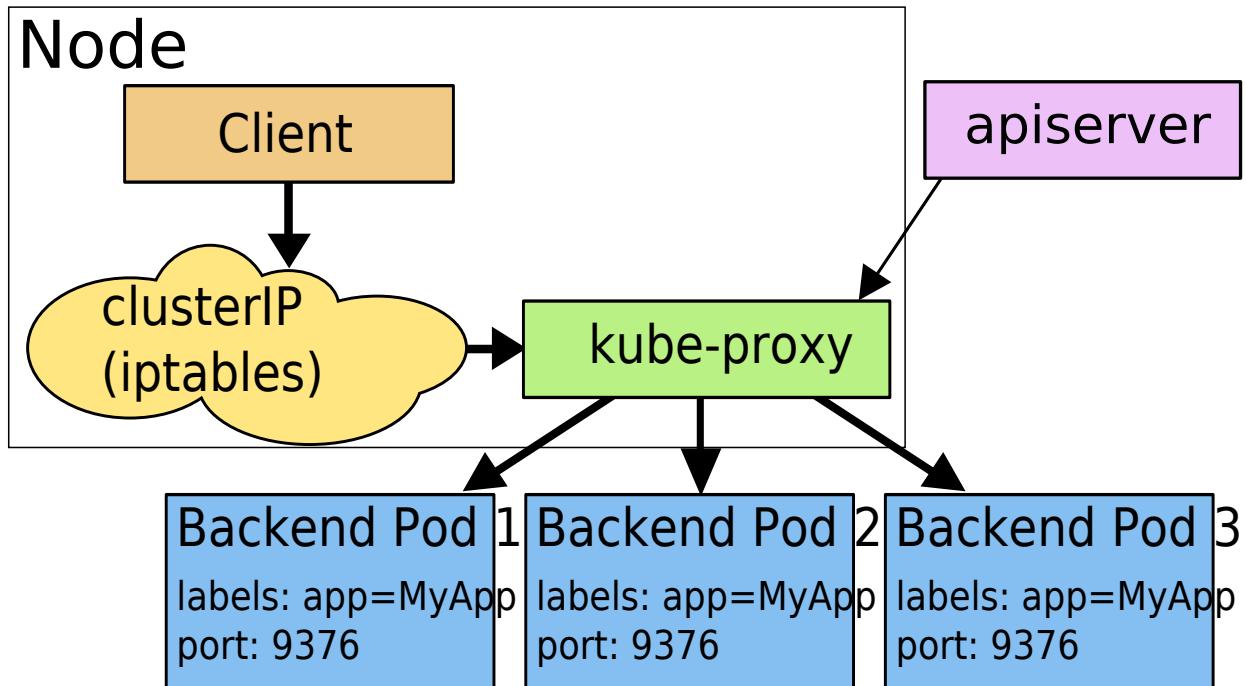
- *There is a long history of DNS implementations not respecting record TTLs, and caching the results of name lookups after they should have expired.*
- *Some apps do DNS lookups only once and cache the results indefinitely.*
- *Even if apps and libraries did proper re-resolution, the low or zero TTLs on the DNS records could impose a high load on DNS that then becomes difficult to manage.*

User space proxy mode

In this mode, kube-proxy watches the Kubernetes master for the addition and removal of Service and Endpoint objects. For each Service it opens a port (randomly chosen) on the local node. Any connections to this "proxy port" are proxied to one of the Service's backend Pods (as reported via Endpoints). kube-proxy takes the `SessionAffinity` setting of the Service into account when deciding which backend Pod to use.

Lastly, the user-space proxy installs iptables rules which capture traffic to the Service's `clusterIP` (which is virtual) and port. The rules redirect that traffic to the proxy port which proxies the backend Pod.

By default, kube-proxy in userspace mode chooses a backend via a round-robin algorithm.



iptables proxy mode

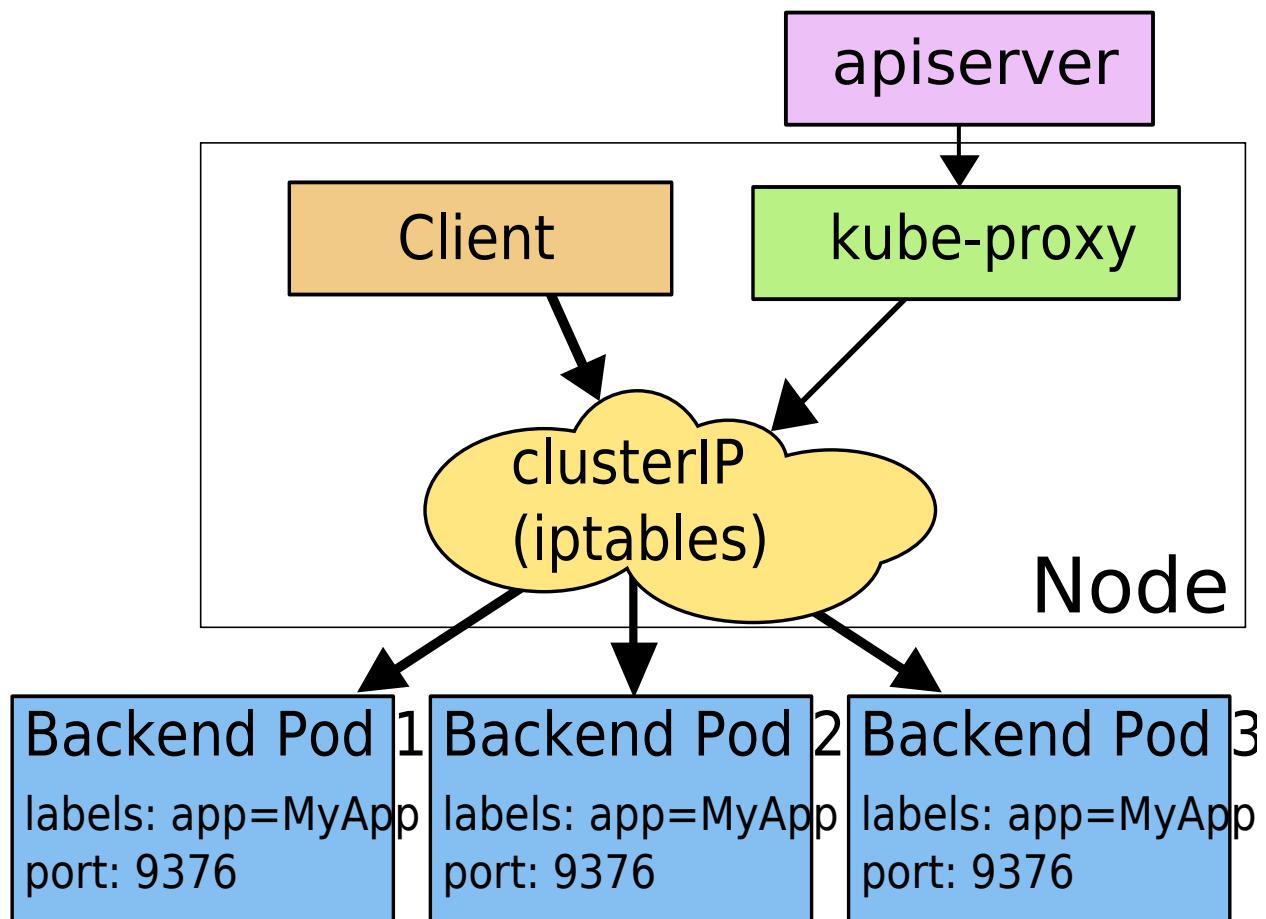
In this mode, kube-proxy watches the Kubernetes control plane for the addition and removal of Service and Endpoint objects. For each Service, it installs iptables rules, which capture traffic to the Service's `clusterIP` and port, and redirect that traffic to one of the Service's backend sets. For each Endpoint object, it installs iptables rules which select a backend Pod.

By default, kube-proxy in iptables mode chooses a backend at random.

Using iptables to handle traffic has a lower system overhead, because traffic is handled by Linux netfilter without the need to switch between userspace and the kernel space. This approach is also likely to be more reliable.

If kube-proxy is running in iptables mode and the first Pod that's selected does not respond, the connection fails. This is different from userspace mode: in that scenario, kube-proxy would detect that the connection to the first Pod had failed and would automatically retry with a different backend Pod.

You can use Pod [readiness probes](#) to verify that backend Pods are working OK, so that kube-proxy in iptables mode only sees backends that test out as healthy. Doing this means you avoid having traffic sent via kube-proxy to a Pod that's known to have failed.



IPVS proxy mode

FEATURE STATE: Kubernetes v1.11 [stable]

In ipvs mode, kube-proxy watches Kubernetes Services and Endpoints, calls netlink interface to create IPVS rules accordingly and synchronizes IPVS

rules with Kubernetes Services and Endpoints periodically. This control loop ensures that IPVS status matches the desired state. When accessing a Service, IPVS directs traffic to one of the backend Pods.

The IPVS proxy mode is based on netfilter hook function that is similar to iptables mode, but uses a hash table as the underlying data structure and works in the kernel space. That means kube-proxy in IPVS mode redirects traffic with lower latency than kube-proxy in iptables mode, with much better performance when synchronising proxy rules. Compared to the other proxy modes, IPVS mode also supports a higher throughput of network traffic.

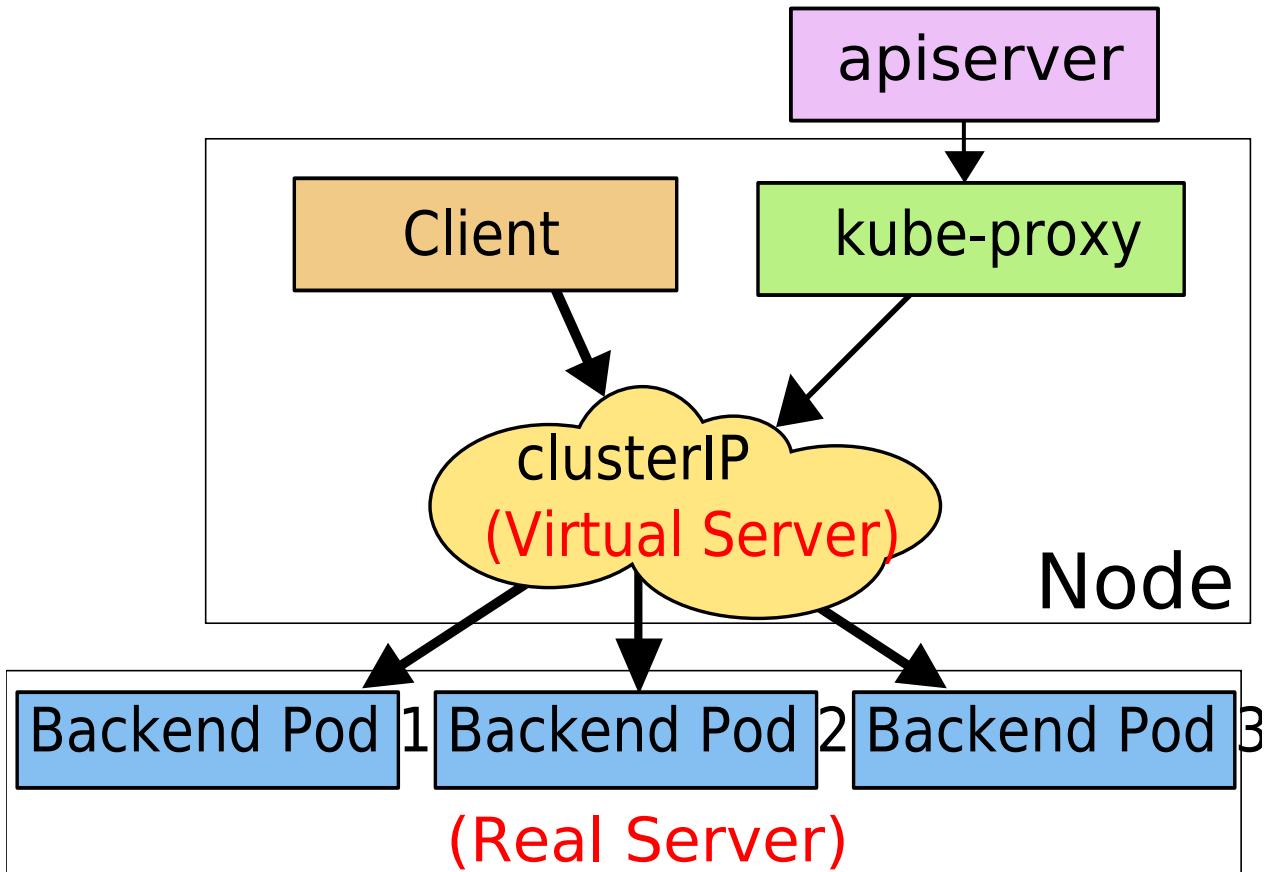
IPVS provides more options for balancing traffic to backend Pods; these are:

- *rr: round-robin*
- *lc: least connection (smallest number of open connections)*
- *dh: destination hashing*
- *sh: source hashing*
- *sed: shortest expected delay*
- *nq: never queue*

Note:

To run kube-proxy in IPVS mode, you must make IPVS available on the node before starting kube-proxy.

When kube-proxy starts in IPVS proxy mode, it verifies whether IPVS kernel modules are available. If the IPVS kernel modules are not detected, then kube-proxy falls back to running in iptables proxy mode.



In these proxy models, the traffic bound for the Service's IP:Port is proxied to an appropriate backend without the clients knowing anything about Kubernetes or Services or Pods.

If you want to make sure that connections from a particular client are passed to the same Pod each time, you can select the session affinity based on the client's IP addresses by setting `service.spec.sessionAffinity` to "ClientIP" (the default is "None"). You can also set the maximum session sticky time by setting `service.spec.sessionAffinityConfig.clientIP.timeoutSeconds` appropriately. (the default value is 10800, which works out to be 3 hours).

Mult-Port Services

For some Services, you need to expose more than one port. Kubernetes lets you configure multiple port definitions on a Service object. When using multiple ports for a Service, you must give all of your ports names so that these are unambiguous. For example:

```
apiVersion: v1
kind: Service
```

```
metadata:  
  name: my-service  
spec:  
  selector:  
    app: MyApp  
  ports:  
    - name: http  
      protocol: TCP  
      port: 80  
      targetPort: 9376  
    - name: https  
      protocol: TCP  
      port: 443  
      targetPort: 9377
```

Note:

As with Kubernetes [names](#) in general, names for ports must only contain lowercase alphanumeric characters and -. Port names must also start and end with an alphanumeric character.

For example, the names 123-abc and web are valid, but 123_abc and -web are not.

Choosing your own IP address

You can specify your own cluster IP address as part of a Service creation request. To do this, set the .spec.clusterIP field. For example, if you already have an existing DNS entry that you wish to reuse, or legacy systems that are configured for a specific IP address and difficult to re-configure.

The IP address that you choose must be a valid IPv4 or IPv6 address from within the service-cluster-ip-range CIDR range that is configured for the API server. If you try to create a Service with an invalid clusterIP address value, the API server will return a 422 HTTP status code to indicate that there's a problem.

Discovering services

Kubernetes supports 2 primary modes of finding a Service - environment variables and DNS.

Environment variables

When a Pod is run on a Node, the kubelet adds a set of environment variables for each active Service. It supports both [Docker links compatible](#) variables (see [makeLinkVariables](#)) and simpler `{SVCNAME}_SERVICE_HOST` and `{SVCNAME}_SERVICE_PORT` variables, where the Service name is upper-cased and dashes are converted to underscores.

For example, the Service `redis-master` which exposes TCP port 6379 and has been allocated cluster IP address 10.0.0.11, produces the following environment variables:

```
REDIS_MASTER_SERVICE_HOST=10.0.0.11
REDIS_MASTER_SERVICE_PORT=6379
REDIS_MASTER_PORT=tcp://10.0.0.11:6379
REDIS_MASTER_PORT_6379_TCP=tcp://10.0.0.11:6379
REDIS_MASTER_PORT_6379_TCP_PROTO=tcp
REDIS_MASTER_PORT_6379_TCP_PORT=6379
REDIS_MASTER_PORT_6379_TCP_ADDR=10.0.0.11
```

Note:

When you have a Pod that needs to access a Service, and you are using the environment variable method to publish the port and cluster IP to the client Pods, you must create the Service before the client Pods come into existence. Otherwise, those client Pods won't have their environment variables populated.

If you only use DNS to discover the cluster IP for a Service, you don't need to worry about this ordering issue.

DNS

You can (and almost always should) set up a DNS service for your Kubernetes cluster using an [add-on](#).

A cluster-aware DNS server, such as CoreDNS, watches the Kubernetes API for new Services and creates a set of DNS records for each one. If DNS has been enabled throughout your cluster then all Pods should automatically be able to resolve Services by their DNS name.

For example, if you have a Service called `my-service` in a Kubernetes namespace `my-ns`, the control plane and the DNS Service acting together create a DNS record for `my-service.my-ns`. Pods in the `my-ns` namespace should be able to find it by simply doing a name lookup for `my-service` (`my-service.my-ns` would also work).

Pods in other namespaces must qualify the name as `my-service.my-ns`. These names will resolve to the cluster IP assigned for the Service.

Kubernetes also supports DNS SRV (Service) records for named ports. If the `my-service.my-ns` Service has a port named `http` with the protocol set to TCP, you can do a DNS SRV query for `_http._tcp.my-service.my-ns` to discover the port number for `http`, as well as the IP address.

The Kubernetes DNS server is the only way to access `ExternalName` Services. You can find more information about `ExternalName` resolution in [DNS Pods and Services](#).

Headless Services

Sometimes you don't need load-balancing and a single Service IP. In this case, you can create what are termed "headless" Services, by explicitly specifying "None" for the cluster IP (`.spec.clusterIP`).

You can use a headless Service to interface with other service discovery mechanisms, without being tied to Kubernetes' implementation.

For headless Services, a cluster IP is not allocated, kube-proxy does not handle these Services, and there is no load balancing or proxying done by the platform for them. How DNS is automatically configured depends on whether the Service has selectors defined:

With selectors

For headless Services that define selectors, the endpoints controller creates `Endpoints` records in the API, and modifies the DNS configuration to return records (addresses) that point directly to the Pods backing the Service.

Without selectors

For headless Services that do not define selectors, the endpoints controller does not create `Endpoints` records. However, the DNS system looks for and configures either:

- CNAME records for `ExternalName`-type Services.
- A records for any `Endpoints` that share a name with the Service, for all other types.

Publishing Services (*ServiceTypes*)

For some parts of your application (for example, frontends) you may want to expose a Service onto an external IP address, that's outside of your cluster.

Kubernetes ServiceTypes allow you to specify what kind of Service you want. The default is ClusterIP.

Type values and their behaviors are:

- *ClusterIP: Exposes the Service on a cluster-internal IP. Choosing this value makes the Service only reachable from within the cluster. This is the default ServiceType.*
- *[NodePort](#): Exposes the Service on each Node's IP at a static port (the `NodePort`). A `ClusterIP` Service, to which the `NodePort` Service routes, is automatically created. You'll be able to contact the `NodePort` Service, from outside the cluster, by requesting `<NodeIP>:<NodePort>`.*
- *[LoadBalancer](#): Exposes the Service externally using a cloud provider's load balancer. `NodePort` and `ClusterIP` Services, to which the external load balancer routes, are automatically created.*
- *[ExternalName](#): Maps the Service to the contents of the `externalName` field (e.g. `foo.bar.example.com`), by returning a `CNAME` record with its value. No proxying of any kind is set up.*

Note: You need either `kube-dns` version 1.7 or `CoreDNS` version 0.0.8 or higher to use the `ExternalName` type.

You can also use [Ingress](#) to expose your Service. Ingress is not a Service type, but it acts as the entry point for your cluster. It lets you consolidate your routing rules into a single resource as it can expose multiple services under the same IP address.

Type `NodePort`

If you set the `type` field to `NodePort`, the Kubernetes control plane allocates a port from a range specified by `--service-node-port-range` flag (default: 30000-32767). Each node proxies that port (the same port number on every Node) into your Service. Your Service reports the allocated port in its `.spec.ports[*].nodePort` field.

If you want to specify particular IP(s) to proxy the port, you can set the `--nodeport-addresses` flag in `kube-proxy` to particular IP block(s); this is supported since Kubernetes v1.10. This flag takes a comma-delimited list of IP blocks (e.g. `10.0.0.0/8, 192.0.2.0/25`) to specify IP address ranges that `kube-proxy` should consider as local to this node.

For example, if you start kube-proxy with the `--nodeport-addresses=127.0.0.0/8` flag, kube-proxy only selects the loopback interface for NodePort Services. The default for `--nodeport-addresses` is an empty list. This means that kube-proxy should consider all available network interfaces for NodePort. (That's also compatible with earlier Kubernetes releases).

If you want a specific port number, you can specify a value in the `nodePort` field. The control plane will either allocate you that port or report that the API transaction failed. This means that you need to take care of possible port collisions yourself. You also have to use a valid port number, one that's inside the range configured for NodePort use.

Using a NodePort gives you the freedom to set up your own load balancing solution, to configure environments that are not fully supported by Kubernetes, or even to just expose one or more nodes' IPs directly.

Note that this Service is visible as `<NodeIP>.spec.ports[*].nodePort` and `.spec.clusterIP:spec.ports[*].port`. (If the `--nodeport-addresses` flag in kube-proxy is set, would be filtered NodeIP(s).)

For example:

```
apiVersion: v1
kind: Service
metadata:
  name: my-service
spec:
  type: NodePort
  selector:
    app: MyApp
  ports:
    # By default and for convenience, the `targetPort` is set
    # to the same value as the `port` field.
    - port: 80
      targetPort: 80
      # Optional field
      # By default and for convenience, the Kubernetes control
      # plane will allocate a port from a range (default: 30000-32767)
      nodePort: 30007
```

Type LoadBalancer

On cloud providers which support external load balancers, setting the `type` field to `LoadBalancer` provisions a load balancer for your Service. The actual

creation of the load balancer happens asynchronously, and information about the provisioned balancer is published in the Service's `.status.loadBalancer` field. For example:

```
apiVersion: v1
kind: Service
metadata:
  name: my-service
spec:
  selector:
    app: MyApp
  ports:
    - protocol: TCP
      port: 80
      targetPort: 9376
  clusterIP: 10.0.171.239
  type: LoadBalancer
status:
  loadBalancer:
    ingress:
      - ip: 192.0.2.127
```

Traffic from the external load balancer is directed at the backend Pods. The cloud provider decides how it is load balanced.

Some cloud providers allow you to specify the `loadBalancerIP`. In those cases, the load-balancer is created with the user-specified `loadBalancerIP`. If the `loadBalancerIP` field is not specified, the `loadBalancer` is set up with an ephemeral IP address. If you specify a `loadBalancerIP` but your cloud provider does not support the feature, the `loadBalancerIP` field that you set is ignored.

Note:

On Azure, if you want to use a user-specified public type `loadBalancerIP`, you first need to create a static type public IP address resource. This public IP address resource should be in the same resource group of the other automatically created resources of the cluster. For example, `MC_myResourceGroup_myAKSCluster_eastus`.

*Specify the assigned IP address as `loadBalancerIP`. Ensure that you have updated the `securityGroupName` in the cloud provider configuration file. For information about troubleshooting *Creating LoadBalancerFailed* permission issues see, [Use a static IP address with the Azure Kubernetes Service \(AKS\) load balancer](#) or [CreatingLoadBalancerFailed on AKS cluster with advanced networking](#).*

Load balancers with mixed protocol types

FEATURE STATE: Kubernetes v1.20 [alpha]

By default, for LoadBalancer type of Services, when there is more than one port defined, all ports must have the same protocol, and the protocol must be one which is supported by the cloud provider.

If the feature gate MixedProtocolLBService is enabled for the kube-apiserver it is allowed to use different protocols when there is more than one port defined.

Note: The set of protocols that can be used for LoadBalancer type of Services is still defined by the cloud provider.

Disabling load balancer NodePort allocation

FEATURE STATE: Kubernetes v1.20 [alpha]

Starting in v1.20, you can optionally disable node port allocation for a Service Type=LoadBalancer by setting the field spec.allocateLoadBalancerNodePorts to false. This should only be used for load balancer implementations that route traffic directly to pods as opposed to using node ports. By default, spec.allocateLoadBalancerNodePorts is true and type LoadBalancer Services will continue to allocate node ports. If spec.allocateLoadBalancerNodePorts is set to false on an existing Service with allocated node ports, those node ports will NOT be de-allocated automatically. You must explicitly remove the nodePorts entry in every Service port to de-allocate those node ports. You must enable the ServiceLBNodePortControl feature gate to use this field.

Internal load balancer

In a mixed environment it is sometimes necessary to route traffic from Services inside the same (virtual) network address block.

In a split-horizon DNS environment you would need two Services to be able to route both external and internal traffic to your endpoints.

To set an internal load balancer, add one of the following annotations to your Service depending on the cloud Service provider you're using.

- [Default](#)

- [GCP](#)
- [AWS](#)
- [Azure](#)
- [IBM Cloud](#)
- [OpenStack](#)
- [Baidu Cloud](#)
- [Tencent Cloud](#)
- [Alibaba Cloud](#)

Select one of the tabs.

```
[...]
metadata:
  name: my-service
  annotations:
    cloud.google.com/load-balancer-type: "Internal"
[...]
```

```
[...]
metadata:
  name: my-service
  annotations:
    service.beta.kubernetes.io/aws-load-balancer-internal: "true"
[...]
```

```
[...]
metadata:
  name: my-service
  annotations:
    service.beta.kubernetes.io/azure-load-balancer-internal:
    "true"
[...]
```

```
[...]
metadata:
  name: my-service
  annotations:
    service.kubernetes.io/ibm-load-balancer-cloud-provider-
ip-type: "private"
[...]
```

```
[...]
metadata:
  name: my-service
  annotations:
    service.beta.kubernetes.io/openstack-internal-load-
balancer: "true"
[...]
```

```
[...]
metadata:
```

```

  name: my-service
  annotations:
    service.beta.kubernetes.io/cce-load-balancer-internal-
vpc: "true"
[...]
[...]
metadata:
  annotations:
    service.kubernetes.io/qcloud-loadbalancer-internal-subnetid:
subnet-xxxxxx
[...]
[...]
metadata:
  annotations:
    service.beta.kubernetes.io/alibaba-cloud-loadbalancer-
address-type: "intranet"
[...]

```

TLS support on AWS

For partial TLS / SSL support on clusters running on AWS, you can add three annotations to a LoadBalancer service:

```

metadata:
  name: my-service
  annotations:
    service.beta.kubernetes.io/aws-load-balancer-ssl-cert: arn:aw
s:acm:us-east-1:123456789012:certificate/12345678-1234-1234-1234-
123456789012

```

The first specifies the ARN of the certificate to use. It can be either a certificate from a third party issuer that was uploaded to IAM or one created within AWS Certificate Manager.

```

metadata:
  name: my-service
  annotations:
    service.beta.kubernetes.io/aws-load-balancer-backend-
protocol: (https|http|ssl|tcp)

```

The second annotation specifies which protocol a Pod speaks. For HTTPS and SSL, the ELB expects the Pod to authenticate itself over the encrypted connection, using a certificate.

HTTP and HTTPS selects layer 7 proxying: the ELB terminates the connection with the user, parses headers, and injects the X-Forwarded-For header with the user's IP address (Pods only see the IP address of the ELB at the other end of its connection) when forwarding requests.

TCP and SSL selects layer 4 proxying: the ELB forwards traffic without modifying the headers.

In a mixed-use environment where some ports are secured and others are left unencrypted, you can use the following annotations:

```
metadata:  
  name: my-service  
  annotations:  
    service.beta.kubernetes.io/aws-load-balancer-backend-  
protocol: http  
    service.beta.kubernetes.io/aws-load-balancer-ssl-ports: "443,8443"
```

In the above example, if the Service contained three ports, 80, 443, and 8443, then 443 and 8443 would use the SSL certificate, but 80 would just be proxied HTTP.

From Kubernetes v1.9 onwards you can use [predefined AWS SSL policies](#) with HTTPS or SSL listeners for your Services. To see which policies are available for use, you can use the aws command line tool:

```
aws elb describe-load-balancer-policies --query 'PolicyDescriptions[].PolicyName'
```

You can then specify any one of those policies using the "service.beta.kubernetes.io/aws-load-balancer-ssl-negotiation-policy" annotation; for example:

```
metadata:  
  name: my-service  
  annotations:  
    service.beta.kubernetes.io/aws-load-balancer-ssl-  
negotiation-policy: "ELBSecurityPolicy-TLS-1-2-2017-01"
```

PROXY protocol support on AWS

To enable [PROXY protocol](#) support for clusters running on AWS, you can use the following service annotation:

```
metadata:  
  name: my-service  
  annotations:  
    service.beta.kubernetes.io/aws-load-balancer-proxy-  
protocol: "*"
```

Since version 1.3.0, the use of this annotation applies to all ports proxied by the ELB and cannot be configured otherwise.

ELB Access Logs on AWS

There are several annotations to manage access logs for ELB Services on AWS.

The annotation `service.beta.kubernetes.io/aws-load-balancer-access-log-enabled` controls whether access logs are enabled.

The annotation `service.beta.kubernetes.io/aws-load-balancer-access-log-emit-interval` controls the interval in minutes for publishing the access logs. You can specify an interval of either 5 or 60 minutes.

The annotation `service.beta.kubernetes.io/aws-load-balancer-access-log-s3-bucket-name` controls the name of the Amazon S3 bucket where load balancer access logs are stored.

The annotation `service.beta.kubernetes.io/aws-load-balancer-access-log-s3-bucket-prefix` specifies the logical hierarchy you created for your Amazon S3 bucket.

```
metadata:  
  name: my-service  
  annotations:  
    service.beta.kubernetes.io/aws-load-balancer-access-log-  
enabled: "true"  
      # Specifies whether access logs are enabled for the load  
balancer  
      service.beta.kubernetes.io/aws-load-balancer-access-log-  
emit-interval: "60"  
        # The interval for publishing the access logs. You can  
specify an interval of either 5 or 60 (minutes).  
        service.beta.kubernetes.io/aws-load-balancer-access-log-  
s3-bucket-name: "my-bucket"  
          # The name of the Amazon S3 bucket where the access logs  
are stored  
          service.beta.kubernetes.io/aws-load-balancer-access-log-
```

```
s3-bucket-prefix: "my-bucket-prefix/prod"
    # The logical hierarchy you created for your Amazon S3
    bucket, for example `my-bucket-prefix/prod`
```

Connection Draining on AWS

Connection draining for Classic ELBs can be managed with the annotation `service.beta.kubernetes.io/aws-load-balancer-connection-draining-enabled` set to the value of "true". The annotation `service.beta.kubernetes.io/aws-load-balancer-connection-draining-timeout` can also be used to set maximum time, in seconds, to keep the existing connections open before deregistering the instances.

```
metadata:
  name: my-service
  annotations:
    service.beta.kubernetes.io/aws-load-balancer-connection-
draining-enabled: "true"
    service.beta.kubernetes.io/aws-load-balancer-connection-
draining-timeout: "60"
```

Other ELB annotations

There are other annotations to manage Classic Elastic Load Balancers that are described below.

```
metadata:
  name: my-service
  annotations:
    service.beta.kubernetes.io/aws-load-balancer-connection-
idle-timeout: "60"
        # The time, in seconds, that the connection is allowed
        to be idle (no data has been sent over the connection) before it
        is closed by the load balancer

    service.beta.kubernetes.io/aws-load-balancer-cross-zone-
load-balancing-enabled: "true"
        # Specifies whether cross-zone load balancing is enabled
        for the load balancer

    service.beta.kubernetes.io/aws-load-balancer-additional-
resource-tags: "environment=prod,owner=devops"
        # A comma-separated list of key-value pairs which will
        be recorded as
        # additional tags in the ELB.
```

```

    service.beta.kubernetes.io/aws-load-balancer-healthcheck-
healthy-threshold: ""
        # The number of successive successful health checks
        required for a backend to
            # be considered healthy for traffic. Defaults to 2, must
            be between 2 and 10

    service.beta.kubernetes.io/aws-load-balancer-healthcheck-
unhealthy-threshold: "3"
        # The number of unsuccessful health checks required for
        a backend to be
            # considered unhealthy for traffic. Defaults to 6, must
            be between 2 and 10

    service.beta.kubernetes.io/aws-load-balancer-healthcheck-
interval: "20"
        # The approximate interval, in seconds, between health
        checks of an
            # individual instance. Defaults to 10, must be between 5
            and 300

    service.beta.kubernetes.io/aws-load-balancer-healthcheck-
timeout: "5"
        # The amount of time, in seconds, during which no
        response means a failed
            # health check. This value must be less than the
service.beta.kubernetes.io/aws-load-balancer-healthcheck-interval
            # value. Defaults to 5, must be between 2 and 60

    service.beta.kubernetes.io/aws-load-balancer-security-
groups: "sg-53fae93f"
        # A list of existing security groups to be added to ELB
        created. Unlike the annotation
            # service.beta.kubernetes.io/aws-load-balancer-extra-
            # security-groups, this replaces all other security groups
            previously assigned to the ELB.

    service.beta.kubernetes.io/aws-load-balancer-extra-
security-groups: "sg-53fae93f,sg-42efd82e"
        # A list of additional security groups to be added to
        the ELB

    service.beta.kubernetes.io/aws-load-balancer-target-node-
labels: "ingress-gw,gw-name=public-api"
        # A comma separated list of key-value pairs which are
        used
            # to select the target nodes for the load balancer

```

Network Load Balancer support on AWS

FEATURE STATE: Kubernetes v1.15 [beta]

To use a Network Load Balancer on AWS, use the annotation `service.beta.kubernetes.io/aws-load-balancer-type` with the value set to `nlb`.

```
metadata:  
  name: my-service  
  annotations:  
    service.beta.kubernetes.io/aws-load-balancer-type: "nlb"
```

Note: NLB only works with certain instance classes; see the [AWS documentation](#) on Elastic Load Balancing for a list of supported instance types.

Unlike Classic Elastic Load Balancers, Network Load Balancers (NLBs) forward the client's IP address through to the node. If a Service's `.spec.externalTrafficPolicy` is set to `Cluster`, the client's IP address is not propagated to the end Pods.

By setting `.spec.externalTrafficPolicy` to `Local`, the client IP addresses is propagated to the end Pods, but this could result in uneven distribution of traffic. Nodes without any Pods for a particular LoadBalancer Service will fail the NLB Target Group's health check on the auto-assigned `.spec.healthCheckNodePort` and not receive any traffic.

In order to achieve even traffic, either use a DaemonSet or specify a [pod anti-affinity](#) to not locate on the same node.

You can also use NLB Services with the [internal load balancer](#) annotation.

In order for client traffic to reach instances behind an NLB, the Node security groups are modified with the following IP rules:

Rule	Protocol	Port(s)	IpRange(s)
<code>Health Check</code>	TCP	<code>NodePort(s)</code> (<code>.spec.healthCheckNodePort</code> for <code>.spec.externalTrafficPolicy = Local</code>)	VPC CIDR
<code>Client Traffic</code>	TCP	<code>NodePort(s)</code>	<code>.spec.loadBalancerSourceRanges</code> (defaults to <code>0.0.0.0/0</code>)
<code>MTU Discovery</code>	ICMP	3,4	<code>.spec.loadBalancerSourceRanges</code> (defaults to <code>0.0.0.0/0</code>)

In order to limit which client IP's can access the Network Load Balancer, specify `loadBalancerSourceRanges`.

```
spec:  
  loadBalancerSourceRanges:  
    - "143.231.0.0/16"
```

Note: If `.spec.loadBalancerSourceRanges` is not set, Kubernetes allows traffic from `0.0.0.0/0` to the Node Security Group(s). If nodes have public IP addresses, be aware that non-NLB traffic can also reach all instances in those modified security groups.

Other CLB annotations on Tencent Kubernetes Engine (TKE)

There are other annotations for managing Cloud Load Balancers on TKE as shown below.

```
metadata:  
  name: my-service  
  annotations:  
    # Bind Loadbalancers with specified nodes  
    service.kubernetes.io/qcloud-loadbalancer-backends-  
label: key in (value1, value2)  
  
    # ID of an existing load balancer  
    service.kubernetes.io/tke-existed-lbidišlb-6swtxxxx  
  
    # Custom parameters for the load balancer (LB), does not  
    support modification of LB type yet  
    service.kubernetes.io/service.extensiveParameters: ""  
  
    # Custom parameters for the LB listener  
    service.kubernetes.io/service.listenerParameters: ""  
  
    # Specifies the type of Load balancer;  
    # valid values: classic (Classic Cloud Load Balancer) or  
    application (Application Cloud Load Balancer)  
    service.kubernetes.io/loadbalance-type: xxxxx  
  
    # Specifies the public network bandwidth billing method;  
    # valid values: TRAFFIC_POSTPAID_BY_HOUR(bill-by-  
    traffic) and BANDWIDTH_POSTPAID_BY_HOUR (bill-by-bandwidth).  
    service.kubernetes.io/qcloud-loadbalancer-internet-  
charge-type: xxxxxx  
  
      # Specifies the bandwidth value (value range: [1,2000]  
Mbps).  
      service.kubernetes.io/qcloud-loadbalancer-internet-max-  
bandwidth-out: "10"  
  
      # When this annotation is set the loadbalancers will
```

```
only register nodes
    # with pod running on it, otherwise all nodes will be
registered.
    service.kubernetes.io/local-svc-only-bind-node-with-pod:
true
```

Type ExternalName

Services of type `ExternalName` map a Service to a DNS name, not to a typical selector such as `my-service` or `cassandra`. You specify these Services with the `spec.externalName` parameter.

This Service definition, for example, maps the `my-service` Service in the `prod` namespace to `my.database.example.com`:

```
apiVersion: v1
kind: Service
metadata:
  name: my-service
  namespace: prod
spec:
  type: ExternalName
  externalName: my.database.example.com
```

Note: `ExternalName` accepts an IPv4 address string, but as a DNS names comprised of digits, not as an IP address. `ExternalNames` that resemble IPv4 addresses are not resolved by CoreDNS or `ingress-nginx` because `ExternalName` is intended to specify a canonical DNS name. To hardcode an IP address, consider using [headless Services](#).

When looking up the host `my-service.prod.svc.cluster.local`, the cluster DNS Service returns a `CNAME` record with the value `my.database.example.com`. Accessing `my-service` works in the same way as other Services but with the crucial difference that redirection happens at the DNS level rather than via proxying or forwarding. Should you later decide to move your database into your cluster, you can start its Pods, add appropriate selectors or endpoints, and change the Service's type.

Warning:

You may have trouble using `ExternalName` for some common protocols, including `HTTP` and `HTTPS`. If you use `ExternalName` then the hostname used by clients inside your cluster is different from the name that the `ExternalName` references.

For protocols that use hostnames this difference may lead to errors or unexpected responses. HTTP requests will have a `Host`: header that the origin server does not recognize; TLS servers will not be able to provide a certificate matching the hostname that the client connected to.

Note: This section is indebted to the [Kubernetes Tips - Part 1](#) blog post from [Alen Komljen](#).

External IPs

If there are external IPs that route to one or more cluster nodes, Kubernetes Services can be exposed on those external IPs. Traffic that ingresses into the cluster with the external IP (as destination IP), on the Service port, will be routed to one of the Service endpoints. externalIPs are not managed by Kubernetes and are the responsibility of the cluster administrator.

In the Service spec, externalIPs can be specified along with any of the ServiceTypes. In the example below, "my-service" can be accessed by clients on "80.11.12.10:80" (externalIP:port)

```
apiVersion: v1
kind: Service
metadata:
  name: my-service
spec:
  selector:
    app: MyApp
  ports:
    - name: http
      protocol: TCP
      port: 80
      targetPort: 9376
  externalIPs:
    - 80.11.12.10
```

Shortcomings

Using the userspace proxy for VIPs works at small to medium scale, but will not scale to very large clusters with thousands of Services. The [original design proposal for portals](#) has more details on this.

Using the userspace proxy obscures the source IP address of a packet accessing a Service. This makes some kinds of network filtering (firewalling)

impossible. The iptables proxy mode does not obscure in-cluster source IPs, but it does still impact clients coming through a load balancer or node-port.

The Type field is designed as nested functionality - each level adds to the previous. This is not strictly required on all cloud providers (e.g. Google Compute Engine does not need to allocate a NodePort to make LoadBalance r work, but AWS does) but the current API requires it.

Virtual IP implementation

The previous information should be sufficient for many people who just want to use Services. However, there is a lot going on behind the scenes that may be worth understanding.

Avoiding collisions

One of the primary philosophies of Kubernetes is that you should not be exposed to situations that could cause your actions to fail through no fault of your own. For the design of the Service resource, this means not making you choose your own port number if that choice might collide with someone else's choice. That is an isolation failure.

In order to allow you to choose a port number for your Services, we must ensure that no two Services can collide. Kubernetes does that by allocating each Service its own IP address.

To ensure each Service receives a unique IP, an internal allocator atomically updates a global allocation map in [etcd](#) prior to creating each Service. The map object must exist in the registry for Services to get IP address assignments, otherwise creations will fail with a message indicating an IP address could not be allocated.

In the control plane, a background controller is responsible for creating that map (needed to support migrating from older versions of Kubernetes that used in-memory locking). Kubernetes also uses controllers to check for invalid assignments (eg due to administrator intervention) and for cleaning up allocated IP addresses that are no longer used by any Services.

Service IP addresses

Unlike Pod IP addresses, which actually route to a fixed destination, Service IPs are not actually answered by a single host. Instead, kube-proxy uses iptables (packet processing logic in Linux) to define virtual IP addresses

which are transparently redirected as needed. When clients connect to the VIP, their traffic is automatically transported to an appropriate endpoint. The environment variables and DNS for Services are actually populated in terms of the Service's virtual IP address (and port).

kube-proxy supports three proxy modes—userspace, iptables and IPVS—which each operate slightly differently.

Userspace

As an example, consider the image processing application described above. When the backend Service is created, the Kubernetes master assigns a virtual IP address, for example 10.0.0.1. Assuming the Service port is 1234, the Service is observed by all of the kube-proxy instances in the cluster. When a proxy sees a new Service, it opens a new random port, establishes an iptables redirect from the virtual IP address to this new port, and starts accepting connections on it.

When a client connects to the Service's virtual IP address, the iptables rule kicks in, and redirects the packets to the proxy's own port. The "Service proxy" chooses a backend, and starts proxying traffic from the client to the backend.

This means that Service owners can choose any port they want without risk of collision. Clients can simply connect to an IP and port, without being aware of which Pods they are actually accessing.

iptables

Again, consider the image processing application described above. When the backend Service is created, the Kubernetes control plane assigns a virtual IP address, for example 10.0.0.1. Assuming the Service port is 1234, the Service is observed by all of the kube-proxy instances in the cluster. When a proxy sees a new Service, it installs a series of iptables rules which redirect from the virtual IP address to per-Service rules. The per-Service rules link to per-Endpoint rules which redirect traffic (using destination NAT) to the backends.

When a client connects to the Service's virtual IP address the iptables rule kicks in. A backend is chosen (either based on session affinity or randomly) and packets are redirected to the backend. Unlike the userspace proxy, packets are never copied to userspace, the kube-proxy does not have to be running for the virtual IP address to work, and Nodes see traffic arriving from the unaltered client IP address.

This same basic flow executes when traffic comes in through a node-port or through a load-balancer, though in those cases the client IP does get altered.

IPVS

iptables operations slow down dramatically in large scale cluster e.g 10,000 Services. IPVS is designed for load balancing and based on in-kernel hash tables. So you can achieve performance consistency in large number of Services from IPVS-based kube-proxy. Meanwhile, IPVS-based kube-proxy has more sophisticated load balancing algorithms (least conn, locality, weighted, persistence).

API Object

Service is a top-level resource in the Kubernetes REST API. You can find more details about the API object at: [Service API object](#).

Supported protocols

TCP

You can use TCP for any kind of Service, and it's the default network protocol.

UDP

You can use UDP for most Services. For type=LoadBalancer Services, UDP support depends on the cloud provider offering this facility.

SCTP

FEATURE STATE: Kubernetes v1.20 [stable]

When using a network plugin that supports SCTP traffic, you can use SCTP for most Services. For type=LoadBalancer Services, SCTP support depends on the cloud provider offering this facility. (Most do not).

Warnings

Support for multihomed SCTP associations

Warning:

The support of multihomed SCTP associations requires that the CNI plugin can support the assignment of multiple interfaces and IP addresses to a Pod.

NAT for multihomed SCTP associations requires special logic in the corresponding kernel modules.

Windows

Note: *SCTP is not supported on Windows based nodes.*

Userspace kube-proxy

Warning: *The kube-proxy does not support the management of SCTP associations when it is in userspace mode.*

HTTP

If your cloud provider supports it, you can use a Service in LoadBalancer mode to set up external HTTP / HTTPS reverse proxying, forwarded to the Endpoints of the Service.

Note: *You can also use [Ingress](#) in place of Service to expose HTTP/ HTTPS Services.*

PROXY protocol

If your cloud provider supports it, you can use a Service in LoadBalancer mode to configure a load balancer outside of Kubernetes itself, that will forward connections prefixed with [PROXY protocol](#).

The load balancer will send an initial series of octets describing the incoming connection, similar to this example

```
PROXY TCP4 192.0.2.202 10.0.42.7 12345 7\r\n
```

followed by the data from the client.

What's next

- Read [Connecting Applications with Services](#)
- Read about [Ingress](#)
- Read about [EndpointSlices](#)

Feedback

Was this page helpful?

Yes No

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

Last modified November 24, 2020 at 9:38 PM PST: [add docs for service.spec.allocateLoadBalancerNodePorts \(acb476bec\)](#)
[Edit this page](#) [Create child page](#) [Create an issue](#)

- [Motivation](#)
- [Service resources](#)
 - [Cloud-native service discovery](#)
- [Defining a Service](#)
 - [Services without selectors](#)
 - [EndpointSlices](#)
 - [Application protocol](#)
- [Virtual IPs and service proxies](#)
 - [Why not use round-robin DNS?](#)
 - [User space proxy mode](#)
 - [iptables proxy mode](#)
 - [IPVS proxy mode](#)
- [Multi-Port Services](#)
- [Choosing your own IP address](#)
- [Discovering services](#)
 - [Environment variables](#)
 - [DNS](#)
- [Headless Services](#)
 - [With selectors](#)
 - [Without selectors](#)
- [Publishing Services \(ServiceTypes\)](#)
 - [Type NodePort](#)
 - [Type LoadBalancer](#)
 - [Type ExternalName](#)
 - [External IPs](#)
- [Shortcomings](#)
- [Virtual IP implementation](#)
 - [Avoiding collisions](#)

- [Service IP addresses](#)
- [API Object](#)
- [Supported protocols](#)
 - [TCP](#)
 - [UDP](#)
 - [SCTP](#)
 - [HTTP](#)
 - [PROXY protocol](#)
- [What's next](#)

Service Topology

FEATURE STATE: Kubernetes v1.17 [alpha]

Service Topology enables a service to route traffic based upon the Node topology of the cluster. For example, a service can specify that traffic be preferentially routed to endpoints that are on the same Node as the client, or in the same availability zone.

Introduction

By default, traffic sent to a ClusterIP or NodePort Service may be routed to any backend address for the Service. Since Kubernetes 1.7 it has been possible to route "external" traffic to the Pods running on the Node that received the traffic, but this is not supported for ClusterIP Services, and more complex topologies – such as routing zonally – have not been possible. The Service Topology feature resolves this by allowing the Service creator to define a policy for routing traffic based upon the Node labels for the originating and destination Nodes.

By using Node label matching between the source and destination, the operator may designate groups of Nodes that are "closer" and "farther" from one another, using whatever metric makes sense for that operator's requirements. For many operators in public clouds, for example, there is a preference to keep service traffic within the same zone, because interzonal traffic has a cost associated with it, while intrazonal traffic does not. Other common needs include being able to route traffic to a local Pod managed by a DaemonSet, or keeping traffic to Nodes connected to the same top-of-rack switch for the lowest latency.

Using Service Topology

If your cluster has Service Topology enabled, you can control Service traffic routing by specifying the topologyKeys field on the Service spec. This field is a preference-order list of Node labels which will be used to sort endpoints when accessing this Service. Traffic will be directed to a Node whose value for the first label matches the originating Node's value for that label. If there is no backend for the Service on a matching Node, then the second label will be considered, and so forth, until no labels remain.

If no match is found, the traffic will be rejected, just as if there were no backends for the Service at all. That is, endpoints are chosen based on the first topology key with available backends. If this field is specified and all entries have no backends that match the topology of the client, the service has no backends for that client and connections should fail. The special value "" may be used to mean "any topology". This catch-all value, if used, only makes sense as the last value in the list.*

If topologyKeys is not specified or empty, no topology constraints will be applied.

Consider a cluster with Nodes that are labeled with their hostname, zone name, and region name. Then you can set the topologyKeys values of a service to direct traffic as follows.

- Only to endpoints on the same node, failing if no endpoint exists on the node: `["kubernetes.io/hostname"]`.
- Preferentially to endpoints on the same node, falling back to endpoints in the same zone, followed by the same region, and failing otherwise: `["kubernetes.io/hostname", "topology.kubernetes.io/zone", "topology.kubernetes.io/region"]`. This may be useful, for example, in cases where data locality is critical.
- Preferentially to the same zone, but fallback on any available endpoint if none are available within this zone: `["topology.kubernetes.io/zone", "*"]`.

Constraints

- Service topology is not compatible with `externalTrafficPolicy=Local`, and therefore a Service cannot use both of these features. It is possible to use both features in the same cluster on different Services, just not on the same Service.
- Valid topology keys are currently limited to `kubernetes.io/hostname`, `topology.kubernetes.io/zone`, and `topology.kubernetes.io/region`, but will be generalized to other node labels in the future.
- Topology keys must be valid label keys and at most 16 keys may be specified.
- The catch-all value, `"*"`, must be the last value in the topology keys, if it is used.

Examples

The following are common examples of using the Service Topology feature.

Only Node Local Endpoints

A Service that only routes to node local endpoints. If no endpoints exist on the node, traffic is dropped:

```
apiVersion: v1
kind: Service
metadata:
  name: my-service
spec:
  selector:
    app: my-app
  ports:
    - protocol: TCP
      port: 80
      targetPort: 9376
  topologyKeys:
    - "kubernetes.io/hostname"
```

Prefer Node Local Endpoints

A Service that prefers node local Endpoints but falls back to cluster wide endpoints if node local endpoints do not exist:

```
apiVersion: v1
kind: Service
metadata:
  name: my-service
spec:
  selector:
    app: my-app
  ports:
    - protocol: TCP
      port: 80
      targetPort: 9376
  topologyKeys:
    - "kubernetes.io/hostname"
    - "*"
```

Only Zonal or Regional Endpoints

A Service that prefers zonal then regional endpoints. If no endpoints exist in either, traffic is dropped.

```
apiVersion: v1
kind: Service
metadata:
  name: my-service
spec:
  selector:
    app: my-app
  ports:
    - protocol: TCP
      port: 80
      targetPort: 9376
  topologyKeys:
```

- "topology.kubernetes.io/zone"
- "topology.kubernetes.io/region"

Prefer Node Local, Zonal, then Regional Endpoints

A Service that prefers node local, zonal, then regional endpoints but falls back to cluster wide endpoints.

```
apiVersion: v1
kind: Service
metadata:
  name: my-service
spec:
  selector:
    app: my-app
  ports:
    - protocol: TCP
      port: 80
      targetPort: 9376
  topologyKeys:
    - "kubernetes.io/hostname"
    - "topology.kubernetes.io/zone"
    - "topology.kubernetes.io/region"
    - "*"
```

What's next

- Read about [enabling Service Topology](#)
- Read [Connecting Applications with Services](#)

Feedback

Was this page helpful?

Yes No

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

Last modified May 30, 2020 at 3:10 PM PST: [add en pages \(ecc27bbbe\)](#)
[Edit this page](#) [Create child page](#) [Create an issue](#)

- [Introduction](#)
- [Using Service Topology](#)
- [Constraints](#)
- [Examples](#)
 - [Only Node Local Endpoints](#)
 - [Prefer Node Local Endpoints](#)
 - [Only Zonal or Regional Endpoints](#)
 - [Prefer Node Local, Zonal, then Regional Endpoints](#)

- [What's next](#)

DNS for Services and Pods

This page provides an overview of DNS support by Kubernetes.

Introduction

Kubernetes DNS schedules a DNS Pod and Service on the cluster, and configures the kubelets to tell individual containers to use the DNS Service's IP to resolve DNS names.

What things get DNS names?

Every Service defined in the cluster (including the DNS server itself) is assigned a DNS name. By default, a client Pod's DNS search list will include the Pod's own namespace and the cluster's default domain. This is best illustrated by example:

Assume a Service named `foo` in the Kubernetes namespace `bar`. A Pod running in namespace `bar` can look up this service by simply doing a DNS query for `foo`. A Pod running in namespace `quux` can look up this service by doing a DNS query for `foo.bar`.

The following sections detail the supported record types and layout that is supported. Any other layout or names or queries that happen to work are considered implementation details and are subject to change without warning. For more up-to-date specification, see [Kubernetes DNS-Based Service Discovery](#).

Services

A/AAAA records

"Normal" (not headless) Services are assigned a DNS A or AAAA record, depending on the IP family of the service, for a name of the form `my-svc.my-namespace.svc.cluster-domain.example`. This resolves to the cluster IP of the Service.

"Headless" (without a cluster IP) Services are also assigned a DNS A or AAAA record, depending on the IP family of the service, for a name of the form `my-svc.my-namespace.svc.cluster-domain.example`. Unlike normal Services, this resolves to the set of IPs of the pods selected by the Service. Clients are expected to consume the set or else use standard round-robin selection from the set.

SRV records

SRV Records are created for named ports that are part of normal or [Headless Services](#). For each named port, the SRV record would have the form `_my-port-name._my-port-protocol.my-svc.my-namespace.svc.cluster-domain.example`. For a regular service, this resolves to the port number and the domain name: `my-svc.my-namespace.svc.cluster-domain.example`. For a headless service, this resolves to multiple answers, one for each pod that is backing the service, and contains the port number and the domain name of the pod of the form `auto-generated-name.my-svc.my-namespace.svc.cluster-domain.example`.

Pods

A/AAAA records

In general a pod has the following DNS resolution:

`pod-ip-address.my-namespace.pod.cluster-domain.example`.

For example, if a pod in the default namespace has the IP address 172.17.0.3, and the domain name for your cluster is `cluster.local`, then the Pod has a DNS name:

`172-17-0-3.default.pod.cluster.local`.

Any pods created by a Deployment or DaemonSet exposed by a Service have the following DNS resolution available:

`pod-ip-address.deployment-name.my-namespace.svc.cluster-domain.example`.

Pod's hostname and subdomain fields

Currently when a pod is created, its hostname is the Pod's `metadata.name` value.

The Pod spec has an optional `hostname` field, which can be used to specify the Pod's hostname. When specified, it takes precedence over the Pod's name to be the hostname of the pod. For example, given a Pod with `hostname` set to "my-host", the Pod will have its hostname set to "my-host".

The Pod spec also has an optional `subdomain` field which can be used to specify its subdomain. For example, a Pod with `hostname` set to "foo", and `subdomain` set to "bar", in namespace "my-namespace", will have the fully qualified domain name (FQDN) "foo.bar.my-namespace.svc.cluster-domain.example".

Example:

```

apiVersion: v1
kind: Service
metadata:
  name: default-subdomain
spec:
  selector:
    name: busybox
  clusterIP: None
  ports:
  - name: foo # Actually, no port is needed.
    port: 1234
    targetPort: 1234
---
apiVersion: v1
kind: Pod
metadata:
  name: busybox1
  labels:
    name: busybox
spec:
  hostname: busybox-1
  subdomain: default-subdomain
  containers:
  - image: busybox:1.28
    command:
      - sleep
      - "3600"
    name: busybox
---
apiVersion: v1
kind: Pod
metadata:
  name: busybox2
  labels:
    name: busybox
spec:
  hostname: busybox-2
  subdomain: default-subdomain
  containers:
  - image: busybox:1.28
    command:
      - sleep
      - "3600"
    name: busybox

```

If there exists a headless service in the same namespace as the pod and with the same name as the subdomain, the cluster's DNS Server also returns an A or AAAA record for the Pod's fully qualified hostname. For example, given a Pod with the hostname set to "busybox-1" and the subdomain set to "default-subdomain", and a headless Service named "default-subdomain" in the same namespace, the pod will see its own FQDN as "busybox-1.default-subdomain.my-namespace.svc.cluster-domain.example". DNS serves an A

or AAAA record at that name, pointing to the Pod's IP. Both pods "busybox1" and "busybox2" can have their distinct A or AAAA records.

The Endpoints object can specify the `hostname` for any endpoint addresses, along with its IP.

Note: Because A or AAAA records are not created for Pod names, `hostname` is required for the Pod's A or AAAA record to be created. A Pod with no `hostname` but with `subdomain` will only create the A or AAAA record for the headless service (`default-subdomain.my-namespace.svc.cluster-domain.example`), pointing to the Pod's IP address. Also, Pod needs to become ready in order to have a record unless `publishNotReadyAddresses=True` is set on the Service.

Pod's `setHostnameAsFQDN` field

FEATURE STATE: Kubernetes v1.20 [beta]

When a Pod is configured to have fully qualified domain name (FQDN), its `hostname` is the short hostname. For example, if you have a Pod with the fully qualified domain name `busybox-1.default-subdomain.my-namespace.svc.cluster-domain.example`, then by default the `hostname` command inside that Pod returns `busybox-1` and the `hostname --fqdn` command returns the FQDN.

When you set `setHostnameAsFQDN: true` in the Pod spec, the kubelet writes the Pod's FQDN into the `hostname` for that Pod's namespace. In this case, both `hostname` and `hostname --fqdn` return the Pod's FQDN.

Note:

In Linux, the `hostname` field of the kernel (the `nodename` field of `struct utsname`) is limited to 64 characters.

If a Pod enables this feature and its FQDN is longer than 64 character, it will fail to start. The Pod will remain in Pending status (`ContainerCreating` as seen by `kubectl`) generating error events, such as Failed to construct FQDN from pod hostname and cluster domain, FQDN long-FQDN is too long (64 characters is the max, 70 characters requested). One way of improving user experience for this scenario is to create an [admission webhook controller](#) to control FQDN size when users create top level objects, for example, Deployment.

Pod's DNS Policy

DNS policies can be set on a per-pod basis. Currently Kubernetes supports the following pod-specific DNS policies. These policies are specified in the `dnsPolicy` field of a Pod Spec.

- "Default": The Pod inherits the name resolution configuration from the node that the pods run on. See [related discussion](#) for more details.
- "ClusterFirst": Any DNS query that does not match the configured cluster domain suffix, such as "`www.kubernetes.io`", is forwarded to the upstream nameserver inherited from the node. Cluster administrators may have extra stub-domain and upstream DNS servers configured. See [related discussion](#) for details on how DNS queries are handled in those cases.
- "ClusterFirstWithHostNet": For Pods running with `hostNetwork`, you should explicitly set its DNS policy "ClusterFirstWithHostNet".
- "None": It allows a Pod to ignore DNS settings from the Kubernetes environment. All DNS settings are supposed to be provided using the `dnsConfig` field in the Pod Spec. See [Pod's DNS config](#) subsection below.

Note: "Default" is not the default DNS policy. If `dnsPolicy` is not explicitly specified, then "ClusterFirst" is used.

The example below shows a Pod with its DNS policy set to "ClusterFirstWithHostNet" because it has `hostNetwork` set to `true`.

```
apiVersion: v1
kind: Pod
metadata:
  name: busybox
  namespace: default
spec:
  containers:
    - image: busybox:1.28
      command:
        - sleep
        - "3600"
      imagePullPolicy: IfNotPresent
      name: busybox
  restartPolicy: Always
  hostNetwork: true
  dnsPolicy: ClusterFirstWithHostNet
```

Pod's DNS Config

Pod's DNS Config allows users more control on the DNS settings for a Pod.

The `dnsConfig` field is optional and it can work with any `dnsPolicy` settings. However, when a Pod's `dnsPolicy` is set to "None", the `dnsConfig` field has to be specified.

Below are the properties a user can specify in the dnsConfig field:

- **nameservers**: a list of IP addresses that will be used as DNS servers for the Pod. There can be at most 3 IP addresses specified. When the Pod's dnsPolicy is set to "None", the list must contain at least one IP address, otherwise this property is optional. The servers listed will be combined to the base nameservers generated from the specified DNS policy with duplicate addresses removed.
- **searches**: a list of DNS search domains for hostname lookup in the Pod. This property is optional. When specified, the provided list will be merged into the base search domain names generated from the chosen DNS policy. Duplicate domain names are removed. Kubernetes allows for at most 6 search domains.
- **options**: an optional list of objects where each object may have a name property (required) and a value property (optional). The contents in this property will be merged to the options generated from the specified DNS policy. Duplicate entries are removed.

The following is an example Pod with custom DNS settings:

[service/networking/custom-dns.yaml](#)


```
apiVersion: v1
kind: Pod
metadata:
  namespace: default
  name: dns-example
spec:
  containers:
    - name: test
      image: nginx
  dnsPolicy: "None"
  dnsConfig:
    nameservers:
      - 1.2.3.4
    searches:
      - ns1.svc.cluster-domain.example
      - my.dns.search.suffix
    options:
      - name: ndots
        value: "2"
      - name: edns0
```

When the Pod above is created, the container test gets the following contents in its /etc/resolv.conf file:

```
nameserver 1.2.3.4
search ns1.svc.cluster-domain.example my.dns.search.suffix
options ndots:2 edns0
```

For IPv6 setup, search path and name server should be setup like this:

```
kubectl exec -it dns-example -- cat /etc/resolv.conf
```

The output is similar to this:

```
nameserver fd00:79:30::a
search default.svc.cluster-domain.example svc.cluster-
domain.example cluster-domain.example
options ndots:5
```

Feature availability

The availability of Pod DNS Config and DNS Policy "None" is shown as below.

k8s version	Feature support
1.14	Stable
1.10	Beta (on by default)
1.9	Alpha

What's next

For guidance on administering DNS configurations, check [Configure DNS Service](#)

Feedback

Was this page helpful?

Yes No

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

Last modified October 02, 2020 at 5:59 PM PST: [Updating doc to reflect that setHostnameAsFQDN feature will be beta in v1.20 \(c29185dac\)](#)
[Edit this page](#) [Create child page](#) [Create an issue](#)

- [Introduction](#)
 - [What things get DNS names?](#)
- [Services](#)
 - [A/AAAA records](#)
 - [SRV records](#)
- [Pods](#)
 - [A/AAAA records](#)
 - [Pod's hostname and subdomain fields](#)
 - [Pod's setHostnameAsFQDN field](#)
 - [Pod's DNS Policy](#)
 - [Pod's DNS Config](#)
 - [Feature availability](#)
- [What's next](#)

Connecting Applications with Services

The Kubernetes model for connecting containers

Now that you have a continuously running, replicated application you can expose it on a network. Before discussing the Kubernetes approach to networking, it is worthwhile to contrast it with the "normal" way networking works with Docker.

By default, Docker uses host-private networking, so containers can talk to other containers only if they are on the same machine. In order for Docker containers to communicate across nodes, there must be allocated ports on the machine's own IP address, which are then forwarded or proxied to the containers. This obviously means that containers must either coordinate which ports they use very carefully or ports must be allocated dynamically.

Coordinating port allocations across multiple developers or teams that provide containers is very difficult to do at scale, and exposes users to cluster-level issues outside of their control. Kubernetes assumes that pods can communicate with other pods, regardless of which host they land on. Kubernetes gives every pod its own cluster-private IP address, so you do not need to explicitly create links between pods or map container ports to host ports. This means that containers within a Pod can all reach each other's ports on localhost, and all pods in a cluster can see each other without NAT. The rest of this document elaborates on how you can run reliable services on such a networking model.

This guide uses a simple nginx server to demonstrate proof of concept.

Exposing pods to the cluster

We did this in a previous example, but let's do it once again and focus on the networking perspective. Create an nginx Pod, and note that it has a container port specification:

[service/networking/run-my-nginx.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

```

This makes it accessible from any node in your cluster. Check the nodes the Pod is running on:

```
kubectl apply -f ./run-my-nginx.yaml
kubectl get pods -l run=my-nginx -o wide
```

NAME	READY	STATUS	RESTARTS
AGE	IP	NODE	
my-nginx-3800858182-jr4a2	1/1	Running	0
13s	10.244.3.4	kubernetes-minion-905m	
my-nginx-3800858182-kna2y	1/1	Running	0
13s	10.244.2.5	kubernetes-minion-ljyed	

Check your pods' IPs:

```
kubectl get pods -l run=my-nginx -o yaml | grep podIP
  podIP: 10.244.3.4
  podIP: 10.244.2.5
```

You should be able to ssh into any node in your cluster and curl both IPs. Note that the containers are not using port 80 on the node, nor are there any special NAT rules to route traffic to the pod. This means you can run multiple nginx pods on the same node all using the same containerPort and access them from any other pod or node in your cluster using IP. Like Docker, ports can still be published to the host node's interfaces, but the need for this is radically diminished because of the networking model.

You can read more about [how we achieve this](#) if you're curious.

Creating a Service

So we have pods running nginx in a flat, cluster wide, address space. In theory, you could talk to these pods directly, but what happens when a node dies? The pods die with it, and the Deployment will create new ones, with different IPs. This is the problem a Service solves.

A Kubernetes Service is an abstraction which defines a logical set of Pods running somewhere in your cluster, that all provide the same functionality. When created, each Service is assigned a unique IP address (also called

clusterIP). This address is tied to the lifespan of the Service, and will not change while the Service is alive. Pods can be configured to talk to the Service, and know that communication to the Service will be automatically load-balanced out to some pod that is a member of the Service.

You can create a Service for your 2 nginx replicas with `kubectl expose`:

```
kubectl expose deployment/my-nginx
```

```
service/my-nginx exposed
```

This is equivalent to `kubectl apply -f` the following yaml:

[service/networking/nginx-svc.yaml](#)



```
apiVersion: v1
kind: Service
metadata:
  name: my-nginx
  labels:
    run: my-nginx
spec:
  ports:
  - port: 80
    protocol: TCP
  selector:
    run: my-nginx
```

This specification will create a Service which targets TCP port 80 on any Pod with the `run: my-nginx` label, and expose it on an abstracted Service port (`targetPort`: is the port the container accepts traffic on, `port`: is the abstracted Service port, which can be any port other pods use to access the Service). View [Service API object](#) to see the list of supported fields in service definition. Check your Service:

```
kubectl get svc my-nginx
```

NAME	TYPE	CLUSTER-IP	EXTERNAL-IP	PORT(S)	AGE
my-nginx	ClusterIP	10.0.162.149	<none>	80/TCP	21s

As mentioned previously, a Service is backed by a group of Pods. These Pods are exposed through endpoints. The Service's selector will be evaluated continuously and the results will be POSTed to an Endpoints object also named `my-nginx`. When a Pod dies, it is automatically removed from the endpoints, and new Pods matching the Service's selector will automatically get added to the endpoints. Check the endpoints, and note that the IPs are the same as the Pods created in the first step:

```
kubectl describe svc my-nginx
```

Name:	my-nginx
Namespace:	default

Labels:	run=my-nginx
Annotations:	<none>
Selector:	run=my-nginx
Type:	ClusterIP
IP:	10.0.162.149
Port:	<unset> 80/TCP
Endpoints:	10.244.2.5:80,10.244.3.4:80
Session Affinity:	None
Events:	<none>

```
kubectl get ep my-nginx
```

NAME	ENDPOINTS	AGE
my-nginx	10.244.2.5:80,10.244.3.4:80	1m

You should now be able to curl the nginx Service on <CLUSTER-IP>:<PORT> from any node in your cluster. Note that the Service IP is completely virtual, it never hits the wire. If you're curious about how this works you can read more about the [service proxy](#).

Accessing the Service

Kubernetes supports 2 primary modes of finding a Service - environment variables and DNS. The former works out of the box while the latter requires the [CoreDNS cluster addon](#).

Note: If the service environment variables are not desired (because possible clashing with expected program ones, too many variables to process, only using DNS, etc) you can disable this mode by setting the `enableServiceLinks` flag to `false` on the [pod spec](#).

Environment Variables

When a Pod runs on a Node, the kubelet adds a set of environment variables for each active Service. This introduces an ordering problem. To see why, inspect the environment of your running nginx Pods (your Pod name will be different):

```
kubectl exec my-nginx-3800858182-jr4a2 -- printenv | grep SERVICE
KUBERNETES_SERVICE_HOST=10.0.0.1
KUBERNETES_SERVICE_PORT=443
KUBERNETES_SERVICE_PORT_HTTPS=443
```

Note there's no mention of your Service. This is because you created the replicas before the Service. Another disadvantage of doing this is that the scheduler might put both Pods on the same machine, which will take your entire Service down if it dies. We can do this the right way by killing the 2 Pods and waiting for the Deployment to recreate them. This time around the Service exists before the replicas. This will give you scheduler-level Service

spreading of your Pods (provided all your nodes have equal capacity), as well as the right environment variables:

```
kubectl scale deployment my-nginx --replicas=0; kubectl scale deployment my-nginx --replicas=2;
```

```
kubectl get pods -l run=my-nginx -o wide
```

NAME	READY	STATUS	RESTARTS
AGE	IP	NODE	
my-nginx-3800858182-e9ihh	1/1	Running	0
5s	10.244.2.7	kubernetes-minion-ljyd	
my-nginx-3800858182-j4rm4	1/1	Running	0
5s	10.244.3.8	kubernetes-minion-905m	

You may notice that the pods have different names, since they are killed and recreated.

```
kubectl exec my-nginx-3800858182-e9ihh -- printenv | grep SERVICE
```

```
KUBERNETES_SERVICE_PORT=443
MY_NGINX_SERVICE_HOST=10.0.162.149
KUBERNETES_SERVICE_HOST=10.0.0.1
MY_NGINX_SERVICE_PORT=80
KUBERNETES_SERVICE_PORT_HTTPS=443
```

DNS

Kubernetes offers a DNS cluster addon Service that automatically assigns dns names to other Services. You can check if it's running on your cluster:

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

NAME	TYPE	CLUSTER-IP	EXTERNAL-IP
PORT(S)	AGE		
kube-dns	ClusterIP	10.0.0.10	<none>
TCP 8m			53/UDP, 53/

The rest of this section will assume you have a Service with a long lived IP (my-nginx), and a DNS server that has assigned a name to that IP. Here we use the CoreDNS cluster addon (application name kube-dns), so you can talk to the Service from any pod in your cluster using standard methods (e.g. `gethostbyname()`). If CoreDNS isn't running, you can enable it referring to the [CoreDNS README](#) or [Installing CoreDNS](#). Let's run another curl application to test this:

```
kubectl run curl --image=radial/busyboxplus:curl -i --tty
```

```
Waiting for pod default(curl-131556218-9fnch) to be running,
status is Pending, pod ready: false
Hit enter for command prompt
```

Then, hit enter and run nslookup my-nginx:

```
[ root@curl-131556218-9fnch:/ ]$ nslookup my-nginx
Server:    10.0.0.10
Address 1: 10.0.0.10

Name:      my-nginx
Address 1: 10.0.162.149
```

Securing the Service

Till now we have only accessed the nginx server from within the cluster. Before exposing the Service to the internet, you want to make sure the communication channel is secure. For this, you will need:

- Self signed certificates for https (unless you already have an identity certificate)
- An nginx server configured to use the certificates
- A [secret](#) that makes the certificates accessible to pods

You can acquire all these from the [nginx https example](#). This requires having go and make tools installed. If you don't want to install those, then follow the manual steps later. In short:

```
make keys KEY=/tmp/nginx.key CERT=/tmp/nginx.crt
kubectl create secret tls nginxsecret --key /tmp/nginx.key --cert /tmp/nginx.crt
```

```
secret/nginxsecret created
```

```
kubectl get secrets
```

NAME	TYPE
DATA AGE default-token-il9rc 1 1d	kubernetes.io/service-account-token
nginxsecret 2 1m	kubernetes.io/tls

And also the configmap:

```
kubectl create configmap nginxconfigmap --from-file=default.conf
configmap/nginxconfigmap created
```

```
kubectl get configmaps
```

NAME	DATA	AGE
nginxconfigmap	1	114s

Following are the manual steps to follow in case you run into problems running make (on windows for example):

```
# Create a public private key pair
openssl req -x509 -nodes -days 365 -newkey rsa:2048 -keyout /d/
```

```

tmp/nginx.key -out /d/tmp/nginx.crt -subj "/CN=my-nginx/0=my-
nginx"
# Convert the keys to base64 encoding
cat /d/tmp/nginx.crt | base64
cat /d/tmp/nginx.key | base64

```

Use the output from the previous commands to create a yaml file as follows. The base64 encoded value should all be on a single line.

```

apiVersion: "v1"
kind: "Secret"
metadata:
  name: "nginxsecret"
  namespace: "default"
type: kubernetes.io/tls
data:
  tls.crt: "LS0tLS1CRUdJTiBDRVJUSUZJQ0FURS0tLS0tCK1JSURIekNDQWd1Z
0F3SUJBZ0lKQUp5M3lQK0pzMlpJTUEwR0NTcUdTSWIzRFFFQkJRVUFNQ1l4RVRBUE
JnTlYKQkFNVENHNw5hVzU0YzNaak1SRXdEd1lEVlFRS0V3aHVaMmx1ZUh0M1l6QwV
GdzB4TnpFd01qWXd0ekEzTVRKYQpGdzB4T0RFd01qWXd0ekEzTVRKYU1DWXhFVEFQ
Qmd0VkJBTVRDRzVuYVc1NGMzWmpNUkV3RHdZRFZRUUtFd2h1CloybHVlSE4yWxpDQ
0FTSXdEUVlKS29aSWh2Y05BUUVCQ1FBRGdnRVBBRENDQVFvQ2dnRUJBSjFxSU1SOV
dWM0IKMlZlQ1RMRMtobDRONXl jMEJxYUhIQktMSnJMc8vdzZhU3hRS29GbHlJSU9
4NGUrMlN5ajBFcndCLzlYTnBwbQppeW1CL3JkRldk0Xg5UWhBQUxCZkVaTmNiV3Ns
TVFVcnhBZW50VWt1dk1vLzgvMHRpbGhjc3paenJEYVJ4NEo5Ci82UVRtVVI3a0ZTW
Up0WTVQZkR3cGc3d1VvaDz_mZ1Voam92VG42eHNVR0M2QURV0DBpNXFlZWhNeVI1N2
lmu2YKNHZpaXdIY3hnL3lZR1JBRS9mRTRqakxCdmd0Nj c2SU90S01rZXV3R0l jNDF
hd05tNnNTSzRqYUNGGeGpYSnZaZQp2by9kTlEybHhHWctKT2l3SEhXbXNhGp4WTRa
NVk3R1ZoK0QrWnYvcW1mMFgvbVY0Rmo1NzV3aj FMwVBocWtsCmdhSXZYRyt4U1FVQ
0F3RUFBYU5RTUU0d0hRwURWUjBP0kJZRUZPNG90WkI3YXc10UlsYkROMzhIYkduYn
hFVj cKTUI4R0ExVWRJd1FZTUJhQUZPNG90WkI3YXc10UlsYkROMzhIYkduYnhFVjd
NQXdHQTFVZEV3UUZNQU1CQWY4dwpEUVlKS29aSWh2Y05BUUVGQ1FBRGdnRUJBRVhT
Mw9FU0lFaXdyMDhwCVA0K2NwTHI3Tw5FMTducDBvMm14a1FvCjRGb0RvRjdRZnZqe
E04Tzd2TjB0clcxb2pGSW0vWDE4ZnZaL3k4ZzVaWG40Vm8zc3hKVmRBcStNZC9jTS
tzUGEKNmJjTkNUekZqeFpUV0UrKzE5NS9zb2dm0UZ3VDVDK3U2Q3B5N0M3MTZvUXR
UakViV05VdEt4cXI0Nk10ZwNCMApwRFhwZmdwQTRadkR4NFo3S2RiZDY5eXM30VFH
Ymg5ZW1PZ05NZflsSUswSGt0ejF5WU4vbVpmK3FqTkJqbWzjCkNnMnlwbGQ0Wi8rU
UNQZjl3SkoybFIrY2FnT0R4elBwCgxNSEcybzgvTHFDdnh6elZPUDUxeXdLZEtxaU
MwSVEKQ0I5T2wwlw5scE9UNEh1b2hSUzBPOSt1Mm9KdFzsNUIyczRpbDlhZ3RTVXF
xULU9Ci0tLS0tRU5EIENFU1RJRK1DQVRFLS0tLS0k"
  tls.key: "LS0tLS1CRUdJTiBQUklwQVRFIEtFWS0tLS0tCK1JSUV2UU1CQURBT
kJna3Foa2lH0XcwQkFRRUZBQVNDQktjd2dnU2pBZ0VBQW9JQkFRQ2RhaURFZlZsZH
dkbFIKd1V5eFpJWmVEZWNuTkFhbWh4d1NpeWF5N1Av0E9ta3NVQ3FCWmNpQ0RzZUh
2dGtzbz1CSzhBZi9WemFhWm9zcApnZjYzUlZuZmNmVUlRQUN3WHhHVfHMXJKVEVG
SzhRSA3VkpMcnplUC9Q0UxZcFlYTE0yYzz3MmtjZUNmZitrCKU1bEVlNUJVbUNUV
09UM3c4S1lPNzFLSwVuNEZJWtZMMMDUrc2JGQmd1Z0ExUE5JdWFubm9UTWt1ZTRuMG
4rTDQKb3NCM01ZUDhtQmtRQlAzeE9JNHl3YjREZXUraURyU2pKSHJzQmlIT05Xc0R
adXJFaXVJMmdoY1kxeWIylWHI2UAozVFV0cGNSbC9pVG9zQngxchJHcl4V09HZvdP
eGxZZmcvbWIvNnBuOUYvNWxlQlkrZStjSTlTMkQ0YXBKWUdpCkwxeHzzVwtGQWdNQ
kFBRUNnZ0VBZFhCK0xbk8ySEl0TGo5bWRsb25IUGlHWVzZ294RGQwci9hQ1Zkan
k4dLEKTjIwL3FQWkUxek1yall6Ry9kVGhTMmMwc0QxaTBXSjdwR1lGb0xtdx1WTj1
tY0FXUTM5SjM0VHzau2FFSWzWNg05TE1jUHHNTmFsNjRLMFRVbUFQZytGam9QSF1h

```

```

UUxLOERLOUtNXXNrSE5p0WNzM1Y5ckd6Vw1VZwtBL0RBULBTCL13L2ZjUFBacDRuR
WVBZmI3WTk1R11l1b1p5V21SU3VKdlNyblBESGtUdw1vVlVwdkxMRHRzaG9reUxiTW
VtN3oKMmJzVmpwSW1GTHJqbGtmQXlpNHg0WjJrV3YyMFRrdWtsZU1jaVlMbjk4QWx
iRi9DSmRLM3QraTRoMTVL2ZQegpoTnh3bk9Qd1VTaDR2Q0o3c2Q5TmtEUGJvS2Jn
eVVH0XBYamZhRGR2UVFLQmdRRFFLM01nUkhkQ1pKNVFqZWFKC1FGdXF4cHdnNzhZT
j0yL1Nwen1UYmtGcVFoQWtyczJxWGX1MDZBRzhrZzIzQkswaHkzaE9zSGgxcXRVK3
NHZVAK0WRERHBsUwV00DZsY2FlR3hoc0V0L1R6cEdtNGFKSm5oNzVVAatVGZk9QTDh
PTm1FZ3MxMVRhUldhNzzxe1RyMgphRlpjQ2pWV1g0YnRSTHVwSkgrMjZnY0FhUUtC
Z1FEQmxVSUUzTnNVOFBZEYvL25sQVB5VWs1T31DdWc3dmVyClUycXlrdXFzYnBkS
i9h0DViT1jhM05IVmpVM25uRGpHVHBWaE9JeXg5TEFrc2RwZEFjVmxvcG9H0DhXYk
9lMTAKMUDqbnkySmdDK3JVWUZiRGtpUGx1K09IYnRn0XFYcGJMSHBzUVpsMGhucDB
YSFNYVm9CMUliQndnMGEy0FVadApCbFBtWmc2d1BRS0JnRHVIUVV2SDZHYTNDVUsx
NFdm0FhIcFFnMU16M2VvWTBPQm5iSDRvZUZKZmcraEppSXlnCm9RN3hqwlVR3BiC
3AyblRtcHERQwlSNzdyRVhsdlht0ElVU2FsbkNiRGlKY01Pc29RdFBZNS9NczJMRm
5LQTQKaENmL0pWb2FtZm1nZEN0ZGtFMXNINE9MR21JVHdEbTRpb0dWZGIwMllnbzF
yb2htNUpLMUI3MkpBb0dBUW01UQpHNDhX0TVhL0w1eSt5dCsyZ3YvUHM2VnBvMjZl
TzRNQ3lJazJVem9ZWE9IYnNkODJkaC8xT2sybGdHZlI2K3VuCnc1YytZUXRSThQ
md3MUpbGhFZDBKTWU3cGpUSVpnQWJ0LzVPbn1Dak90VXN2aDJjS2lrQ1Z2dTzsZl
BjNkQKck1lT2ZIaHhxV0RZK2Q1TGN1YSt2NzJ0RkxhenJsSlBsRzl0ZHhrQ2dZRUf
5elIzT3UyMDNRVVV6bUlCRkwZZAp4Wm5XZ0JLSEo3TnNxcGFwB2RjL0d5aGVycjFD
ZzE2MmJaSjJDV2RsZkI0VEdtUjZZdmxTZEFO0FRwUWhFbUtKCnFBLzVzdHdxNwd0W
GVLOVJmMwXK29xNThRNTBxMmk1NVdUTThoSDZhTjlaMT1tZ0FGdE5VdGNqQUx2dF
YxdEYKWSS4WFJkSHJaRnBIwl2NwkVw1VbGc9Ci0tLS0tRU5EIFBSSVZBVEUgS0V
ZLS0tLS0K"

```

Now create the secrets using the file:

```

kubectl apply -f nginxsecrets.yaml
kubectl get secrets

```

NAME	TYPE
DATA AGE default-token-il9rc 1 1d	kubernetes.io/service-account-token
nginxsecret 2 1m	kubernetes.io/tls

Now modify your nginx replicas to start an https server using the certificate in the secret, and the Service, to expose both ports (80 and 443):

[service/networking/nginx-secure-app.yaml](#)

```

apiVersion: v1
kind: Service
metadata:
  name: my-nginx
  labels:
    run: my-nginx
spec:
  type: NodePort
  ports:

```

```

- port: 8080
  targetPort: 80
  protocol: TCP
  name: http
- port: 443
  protocol: TCP
  name: https
selector:
  run: my-nginx

---

apiVersion: apps/v1
kind: Deployment
metadata:
  name: my-nginx
spec:
  selector:
    matchLabels:
      run: my-nginx
  replicas: 1
  template:
    metadata:
      labels:
        run: my-nginx
    spec:
      volumes:
        - name: secret-volume
          secret:
            secretName: nginxsecret
        - name: configmap-volume
          configMap:
            name: nginxconfigmap
      containers:
        - name: nginxhttps
          image: bprashanth/nginxhttps:1.0
          ports:
            - containerPort: 443
            - containerPort: 80
          volumeMounts:
            - mountPath: /etc/nginx/ssl
              name: secret-volume
            - mountPath: /etc/nginx/conf.d
              name: configmap-volume

```

Noteworthy points about the nginx-secure-app manifest:

- It contains both Deployment and Service specification in the same file.
- The [nginx server](#) serves HTTP traffic on port 80 and HTTPS traffic on 443, and nginx Service exposes both ports.
- Each container has access to the keys through a volume mounted at /etc/nginx/ssl. This is setup before the nginx server is started.

```
kubectl delete deployments,svc my-nginx; kubectl create -f ./nginx-secure-app.yaml
```

At this point you can reach the nginx server from any node.

```
kubectl get pods -o yaml | grep -i podip
  podIP: 10.244.3.5
node $ curl -k https://10.244.3.5
...
<h1>Welcome to nginx!</h1>
```

Note how we supplied the `-k` parameter to `curl` in the last step, this is because we don't know anything about the pods running nginx at certificate generation time, so we have to tell `curl` to ignore the CName mismatch. By creating a Service we linked the CName used in the certificate with the actual DNS name used by pods during Service lookup. Let's test this from a pod (the same secret is being reused for simplicity, the pod only needs `nginx.crt` to access the Service):

[service/networking/curlpod.yaml](#)



```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: curl-deployment
spec:
  selector:
    matchLabels:
      app: curlpod
  replicas: 1
  template:
    metadata:
      labels:
        app: curlpod
    spec:
      volumes:
        - name: secret-volume
          secret:
            secretName: nginxsecret
      containers:
        - name: curlpod
          command:
            - sh
            - -c
            - while true; do sleep 1; done
          image: radial/busyboxplus:curl
          volumeMounts:
            - mountPath: /etc/nginx/ssl
              name: secret-volume
```

```
kubectl apply -f ./curlpod.yaml  
kubectl get pods -l app=curlpod
```

NAME	READY	STATUS
RESTARTS	AGE	
curl-deployment-1515033274-1410r	1/1	Running
0	1m	

```
kubectl exec curl-deployment-1515033274-1410r -- curl https://my-nginx --cacert /etc/nginx/ssl/tls.crt
```

```
...  
<title>Welcome to nginx!</title>  
...
```

Exposing the Service

For some parts of your applications you may want to expose a Service onto an external IP address. Kubernetes supports two ways of doing this: NodePorts and LoadBalancers. The Service created in the last section already used NodePort, so your nginx HTTPS replica is ready to serve traffic on the internet if your node has a public IP.

```
kubectl get svc my-nginx -o yaml | grep nodePort -C 5  
uid: 07191fb3-f61a-11e5-8ae5-42010af00002  
spec:  
  clusterIP: 10.0.162.149  
  ports:  
    - name: http  
      nodePort: 31704  
      port: 8080  
      protocol: TCP  
      targetPort: 80  
    - name: https  
      nodePort: 32453  
      port: 443  
      protocol: TCP  
      targetPort: 443  
  selector:  
    run: my-nginx
```

```
kubectl get nodes -o yaml | grep ExternalIP -C 1  
- address: 104.197.41.11  
  type: ExternalIP  
  allocatable:  
--  
- address: 23.251.152.56  
  type: ExternalIP  
  allocatable:  
...  
$ curl https://<EXTERNAL-IP>:<NODE-PORT> -k
```

```
...
<h1>Welcome to nginx!</h1>
```

Let's now recreate the Service to use a cloud load balancer, just change the Type of my-nginx Service from NodePort to LoadBalancer:

```
kubectl edit svc my-nginx
kubectl get svc my-nginx
```

NAME	TYPE	CLUSTER-IP	EXTERNAL-IP
PORT(S)	AGE		
my-nginx	LoadBalancer	10.0.162.149	xx.xxx.xxx.xxx
8080:30163/TCP	21s		

```
curl https://<EXTERNAL-IP> -k
```

```
...
<title>Welcome to nginx!</title>
```

The IP address in the EXTERNAL-IP column is the one that is available on the public internet. The CLUSTER-IP is only available inside your cluster/private cloud network.

Note that on AWS, type LoadBalancer creates an ELB, which uses a (long) hostname, not an IP. It's too long to fit in the standard kubectl get svc output, in fact, so you'll need to do kubectl describe service my-nginx to see it. You'll see something like this:

```
kubectl describe service my-nginx
...
LoadBalancer Ingress:
a320587ffd19711e5a37606cf4a74574-1142138393.us-
east-1.elb.amazonaws.com
...
```

What's next

- Learn more about [Using a Service to Access an Application in a Cluster](#)
- Learn more about [Connecting a Front End to a Back End Using a Service](#)
- Learn more about [Creating an External Load Balancer](#)

Feedback

Was this page helpful?

Yes No

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

Last modified August 05, 2020 at 3:17 AM PST: [Replace special quote characters with normal ones. \(c6a96128c\)](#)
[Edit this page](#) [Create child page](#) [Create an issue](#)

- [The Kubernetes model for connecting containers](#)
- [Exposing pods to the cluster](#)
- [Creating a Service](#)
- [Accessing the Service](#)
 - [Environment Variables](#)
 - [DNS](#)
- [Securing the Service](#)
- [Exposing the Service](#)
- [What's next](#)

EndpointSlices

FEATURE STATE: Kubernetes v1.17 [beta]

EndpointSlices provide a simple way to track network endpoints within a Kubernetes cluster. They offer a more scalable and extensible alternative to Endpoints.

Motivation

The Endpoints API has provided a simple and straightforward way of tracking network endpoints in Kubernetes. Unfortunately as Kubernetes clusters and [Services](#) have grown to handle and send more traffic to more backend Pods, limitations of that original API became more visible. Most notably, those included challenges with scaling to larger numbers of network endpoints.

Since all network endpoints for a Service were stored in a single Endpoints resource, those resources could get quite large. That affected the performance of Kubernetes components (notably the master control plane) and resulted in significant amounts of network traffic and processing when Endpoints changed. EndpointSlices help you mitigate those issues as well as provide an extensible platform for additional features such as topological routing.

EndpointSlice resources

In Kubernetes, an EndpointSlice contains references to a set of network endpoints. The control plane automatically creates EndpointSlices for any Kubernetes Service that has a [selector](#) specified. These EndpointSlices include references to all the Pods that match the Service selector.

EndpointSlices group network endpoints together by unique combinations of protocol, port number, and Service name. The name of a EndpointSlice object must be a valid [DNS subdomain name](#).

As an example, here's a sample `EndpointSlice` resource for the `example` Kubernetes Service.

```
apiVersion: discovery.k8s.io/v1beta1
kind: EndpointSlice
metadata:
  name: example-abc
  labels:
    kubernetes.io/service-name: example
addressType: IPv4
ports:
  - name: http
    protocol: TCP
    port: 80
endpoints:
  - addresses:
      - "10.1.2.3"
    conditions:
      ready: true
    hostname: pod-1
    topology:
      kubernetes.io/hostname: node-1
      topology.kubernetes.io/zone: us-west2-a
```

By default, the control plane creates and manages `EndpointSlices` to have no more than 100 endpoints each. You can configure this with the `--max-endpoints-per-slice` [kube-controller-manager](#) flag, up to a maximum of 1000.

`EndpointSlices` can act as the source of truth for [kube-proxy](#) when it comes to how to route internal traffic. When enabled, they should provide a performance improvement for services with large numbers of endpoints.

Address types

`EndpointSlices` support three address types:

- `IPv4`
- `IPv6`
- `FQDN (Fully Qualified Domain Name)`

Conditions

The `EndpointSlice` API stores conditions about endpoints that may be useful for consumers. The three conditions are `ready`, `serving`, and `terminating`.

Ready

`ready` is a condition that maps to a Pod's `Ready` condition. A running Pod with the `Ready` condition set to `True` should have this `EndpointSlice` condition also set to `true`. For compatibility reasons, `ready` is NEVER `true` when a Pod is terminating. Consumers should refer to the `serving` condition

to inspect the readiness of terminating Pods. The only exception to this rule is for Services with `spec.publishNotReadyAddresses` set to true. Endpoints for these Services will always have the `ready` condition set to true.

Serving

FEATURE STATE: Kubernetes v1.20 [alpha]

serving is identical to the `ready` condition, except it does not account for terminating states. Consumers of the `EndpointSlice` API should check this condition if they care about pod readiness while the pod is also terminating.

Note: Although `serving` is almost identical to `ready`, it was added to prevent break the existing meaning of `ready`. It may be unexpected for existing clients if `ready` could be `true` for terminating endpoints, since historically terminating endpoints were never included in the `Endpoints` or `EndpointSlice` API to begin with. For this reason, `ready` is always `false` for terminating endpoints, and a new condition `serving` was added in v1.20 so that clients can track readiness for terminating pods independent of the existing semantics for `ready`.

Terminating

FEATURE STATE: Kubernetes v1.20 [alpha]

Terminating is a condition that indicates whether an endpoint is terminating. For pods, this is any pod that has a deletion timestamp set.

Topology information

FEATURE STATE: Kubernetes v1.20 [deprecated]

Note: The `topology` field in `EndpointSlices` has been deprecated and will be removed in a future release. A new `nodeName` field will be used instead of setting `kubernetes.io/hostname` in `topology`. It was determined that other topology fields covering zone and region would be better represented as `EndpointSlice` labels that would apply to all endpoints within the `EndpointSlice`.

Each endpoint within an `EndpointSlice` can contain relevant topology information. This is used to indicate where an endpoint is, containing information about the corresponding Node, zone, and region. When the values are available, the control plane sets the following Topology labels for `EndpointSlices`:

- `kubernetes.io/hostname` - The name of the Node this endpoint is on.
- `topology.kubernetes.io/zone` - The zone this endpoint is in.
- `topology.kubernetes.io/region` - The region this endpoint is in.

The values of these labels are derived from resources associated with each endpoint in a slice. The `hostname` label represents the value of the

NodeName field on the corresponding Pod. The zone and region labels represent the value of the labels with the same names on the corresponding Node.

Management

Most often, the control plane (specifically, the endpoint slice controller) creates and manages EndpointSlice objects. There are a variety of other use cases for EndpointSlices, such as service mesh implementations, that could result in other entities or controllers managing additional sets of EndpointSlices.

To ensure that multiple entities can manage EndpointSlices without interfering with each other, Kubernetes defines the [label](#) `endpointslice.kubernetes.io/managed-by`, which indicates the entity managing an EndpointSlice. The endpoint slice controller sets `endpointslice-controller.k8s.io` as the value for this label on all EndpointSlices it manages. Other entities managing EndpointSlices should also set a unique value for this label.

Ownership

In most use cases, EndpointSlices are owned by the Service that the endpoint slice object tracks endpoints for. This ownership is indicated by an owner reference on each EndpointSlice as well as a `kubernetes.io/service-name` label that enables simple lookups of all EndpointSlices belonging to a Service.

EndpointSlice mirroring

In some cases, applications create custom Endpoints resources. To ensure that these applications do not need to concurrently write to both Endpoints and EndpointSlice resources, the cluster's control plane mirrors most Endpoints resources to corresponding EndpointSlices.

The control plane mirrors Endpoints resources unless:

- the Endpoints resource has a `endpointslice.kubernetes.io/skip-mirror` label set to true.
- the Endpoints resource has a `control-plane.alpha.kubernetes.io/leader` annotation.
- the corresponding Service resource does not exist.
- the corresponding Service resource has a non-nil selector.

Individual Endpoints resources may translate into multiple EndpointSlices. This will occur if an Endpoints resource has multiple subsets or includes endpoints with multiple IP families (IPv4 and IPv6). A maximum of 1000 addresses per subset will be mirrored to EndpointSlices.

Distribution of EndpointSlices

Each EndpointSlice has a set of ports that applies to all endpoints within the resource. When named ports are used for a Service, Pods may end up with different target port numbers for the same named port, requiring different EndpointSlices. This is similar to the logic behind how subsets are grouped with Endpoints.

The control plane tries to fill EndpointSlices as full as possible, but does not actively rebalance them. The logic is fairly straightforward:

1. *Iterate through existing EndpointSlices, remove endpoints that are no longer desired and update matching endpoints that have changed.*
2. *Iterate through EndpointSlices that have been modified in the first step and fill them up with any new endpoints needed.*
3. *If there's still new endpoints left to add, try to fit them into a previously unchanged slice and/or create new ones.*

Importantly, the third step prioritizes limiting EndpointSlice updates over a perfectly full distribution of EndpointSlices. As an example, if there are 10 new endpoints to add and 2 EndpointSlices with room for 5 more endpoints each, this approach will create a new EndpointSlice instead of filling up the 2 existing EndpointSlices. In other words, a single EndpointSlice creation is preferable to multiple EndpointSlice updates.

With kube-proxy running on each Node and watching EndpointSlices, every change to an EndpointSlice becomes relatively expensive since it will be transmitted to every Node in the cluster. This approach is intended to limit the number of changes that need to be sent to every Node, even if it may result with multiple EndpointSlices that are not full.

In practice, this less than ideal distribution should be rare. Most changes processed by the EndpointSlice controller will be small enough to fit in an existing EndpointSlice, and if not, a new EndpointSlice is likely going to be necessary soon anyway. Rolling updates of Deployments also provide a natural repacking of EndpointSlices with all Pods and their corresponding endpoints getting replaced.

Duplicate endpoints

Due to the nature of EndpointSlice changes, endpoints may be represented in more than one EndpointSlice at the same time. This naturally occurs as changes to different EndpointSlice objects can arrive at the Kubernetes client watch/cache at different times. Implementations using EndpointSlice must be able to have the endpoint appear in more than one slice. A reference implementation of how to perform endpoint deduplication can be found in the EndpointSliceCache implementation in kube-proxy.

What's next

- Learn about [Enabling EndpointSlices](#)

- Read [Connecting Applications with Services](#)

Feedback

Was this page helpful?

Yes No

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

Last modified November 24, 2020 at 9:37 PM PST: [add docs for EndpointSlice conditions \(d591f7e3b\)](#)

[Edit this page](#) [Create child page](#) [Create an issue](#)

- [Motivation](#)
- [EndpointSlice resources](#)
 - [Address types](#)
 - [Conditions](#)
 - [Topology information](#)
 - [Management](#)
 - [Ownership](#)
 - [EndpointSlice mirroring](#)
 - [Distribution of EndpointSlices](#)
 - [Duplicate endpoints](#)
- [What's next](#)

Ingress

FEATURE STATE: Kubernetes v1.19 [stable]

An API object that manages external access to the services in a cluster, typically HTTP.

Ingress may provide load balancing, SSL termination and name-based virtual hosting.

Terminology

For clarity, this guide defines the following terms:

- **Node:** A worker machine in Kubernetes, part of a cluster.
- **Cluster:** A set of Nodes that run containerized applications managed by Kubernetes. For this example, and in most common Kubernetes deployments, nodes in the cluster are not part of the public internet.
- **Edge router:** A router that enforces the firewall policy for your cluster. This could be a gateway managed by a cloud provider or a physical piece of hardware.

- *Cluster network*: A set of links, logical or physical, that facilitate communication within a cluster according to the Kubernetes networking model.
- *Service*: A Kubernetes [Service](#) that identifies a set of Pods using [label](#) selectors. Unless mentioned otherwise, Services are assumed to have virtual IPs only routable within the cluster network.

What is Ingress?

[Ingress](#) exposes HTTP and HTTPS routes from outside the cluster to [services](#) within the cluster. Traffic routing is controlled by rules defined on the Ingress resource.

Here is a simple example where an Ingress sends all its traffic to one Service:

[JavaScript must be [enabled](#) to view content]

An Ingress may be configured to give Services externally-reachable URLs, load balance traffic, terminate SSL / TLS, and offer name-based virtual hosting. An [Ingress controller](#) is responsible for fulfilling the Ingress, usually with a load balancer, though it may also configure your edge router or additional frontends to help handle the traffic.

An Ingress does not expose arbitrary ports or protocols. Exposing services other than HTTP and HTTPS to the internet typically uses a service of type [Service.Type=NodePort](#) or [Service.Type=LoadBalancer](#).

Prerequisites

You must have an [Ingress controller](#) to satisfy an Ingress. Only creating an Ingress resource has no effect.

You may need to deploy an Ingress controller such as [ingress-nginx](#). You can choose from a number of [Ingress controllers](#).

Ideally, all Ingress controllers should fit the reference specification. In reality, the various Ingress controllers operate slightly differently.

Note: Make sure you review your Ingress controller's documentation to understand the caveats of choosing it.

The Ingress resource

A minimal Ingress resource example:

[service/networking/minimal-ingress.yaml](#)


```
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: minimal-ingress
  annotations:
    nginx.ingress.kubernetes.io/rewrite-target: /
spec:
  rules:
  - http:
      paths:
      - path: /testpath
        pathType: Prefix
        backend:
          service:
            name: test
            port:
              number: 80
```

As with all other Kubernetes resources, an Ingress needs `apiVersion`, `kind`, and `metadata` fields. The name of an Ingress object must be a valid [DNS subdomain name](#). For general information about working with config files, see [deploying applications](#), [configuring containers](#), [managing resources](#).

Ingress frequently uses annotations to configure some options depending on the Ingress controller, an example of which is the [rewrite-target annotation](#). Different [Ingress controller](#) support different annotations. Review the documentation for your choice of Ingress controller to learn which annotations are supported.

The Ingress `spec` has all the information needed to configure a load balancer or proxy server. Most importantly, it contains a list of rules matched against all incoming requests. Ingress resource only supports rules for directing HTTP(S) traffic.

Ingress rules

Each HTTP rule contains the following information:

- An optional host. In this example, no host is specified, so the rule applies to all inbound HTTP traffic through the IP address specified. If a host is provided (for example, foo.bar.com), the rules apply to that host.
- A list of paths (for example, /testpath), each of which has an associated backend defined with a service.name and a service.port.name or service.port.number. Both the host and path must match the content of an incoming request before the load balancer directs traffic to the referenced Service.
- A backend is a combination of Service and port names as described in the [Service doc](#) or a [custom resource backend](#) by way of a [CRD](#). HTTP (and HTTPS) requests to the Ingress that matches the host and path of the rule are sent to the listed backend.

A defaultBackend is often configured in an Ingress controller to service any requests that do not match a path in the spec.

DefaultBackend

An Ingress with no rules sends all traffic to a single default backend. The defaultBackend is conventionally a configuration option of the [Ingress controller](#) and is not specified in your Ingress resources.

If none of the hosts or paths match the HTTP request in the Ingress objects, the traffic is routed to your default backend.

Resource backends

A Resource backend is an ObjectRef to another Kubernetes resource within the same namespace as the Ingress object. A Resource is a mutually exclusive setting with Service, and will fail validation if both are specified. A common usage for a Resource backend is to ingress data to an object storage backend with static assets.

[service/networking/ingress-resource-backend.yaml](#)
□

```
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: ingress-resource-backend
spec:
  defaultBackend:
    resource:
      apiGroup: k8s.example.com
      kind: StorageBucket
```

```

  name: static-assets
rules:
- http:
  paths:
    - path: /icons
      pathType: ImplementationSpecific
      backend:
        resource:
          apiGroup: k8s.example.com
          kind: StorageBucket
          name: icon-assets

```

After creating the Ingress above, you can view it with the following command:

```
kubectl describe ingress ingress-resource-backend
```

```

Name:           ingress-resource-backend
Namespace:      default
Address:
Default backend: APIGroup: k8s.example.com, Kind:
StorageBucket, Name: static-assets
Rules:
  Host      Path  Backends
  ----      ---   -----
  *          /icons  APIGroup: k8s.example.com, Kind:
  StorageBucket, Name: icon-assets
Annotations:  <none>
Events:       <none>

```

Path types

Each path in an Ingress is required to have a corresponding path type. Paths that do not include an explicit pathType will fail validation. There are three supported path types:

- **ImplementationSpecific**: With this path type, matching is up to the IngressClass. Implementations can treat this as a separate pathType or treat it identically to Prefix or Exact path types.
- **Exact**: Matches the URL path exactly and with case sensitivity.
- **Prefix**: Matches based on a URL path prefix split by /. Matching is case sensitive and done on a path element by element basis. A path element refers to the list of labels in the path split by the / separator. A

request is a match for path p if every p is an element-wise prefix of p of the request path.

Note: If the last element of the path is a substring of the last element in request path, it is not a match (for example: /foo/bar matches /foo/bar/baz, but does not match /foo/barbaz).

Examples

Kind	Path(s)	Request path(s)	Matches?
Prefix	/	(all paths)	Yes
Exact	/foo	/foo	Yes
Exact	/foo	/bar	No
Exact	/foo	/foo/	No
Exact	/foo/	/foo	No
Prefix	/foo	/foo, /foo/	Yes
Prefix	/foo/	/foo, /foo/	Yes
Prefix	/aaa/bb	/aaa/bbb	No
Prefix	/aaa/bbb	/aaa/bbb	Yes
Prefix	/aaa/bbb/	/aaa/bbb	Yes, ignores trailing slash
Prefix	/aaa/bbb	/aaa/bbb/	Yes, matches trailing slash
Prefix	/aaa/bbb	/aaa/bbb/ccc	Yes, matches subpath
Prefix	/aaa/bbb	/aaa/bbbxyz	No, does not match string prefix
Prefix	/, /aaa	/aaa/ccc	Yes, matches /aaa prefix
Prefix	/, /aaa, /aaa/bbb	/aaa/bbb	Yes, matches /aaa/bbb prefix
Prefix	/, /aaa, /aaa/bbb	/ccc	Yes, matches / prefix
Prefix	/aaa	/ccc	No, uses default backend
Mixed	/foo (Prefix), /foo (Exact)	/foo	Yes, prefers Exact

Multiple matches

In some cases, multiple paths within an Ingress will match a request. In those cases precedence will be given first to the longest matching path. If two paths are still equally matched, precedence will be given to paths with an exact path type over prefix path type.

Hostname wildcards

Hosts can be precise matches (for example "foo.bar.com") or a wildcard (for example "*.foo.com"). Precise matches require that the HTTP host

header matches the `host` field. Wildcard matches require the HTTP host header is equal to the suffix of the wildcard rule.

Host	Host header	Match?
<code>*.foo.com</code>	<code>bar.foo.com</code>	Matches based on shared suffix
<code>*.foo.com</code>	<code>baz.bar.foo.com</code>	No match, wildcard only covers a single DNS label
<code>*.foo.com</code>	<code>foo.com</code>	No match, wildcard only covers a single DNS label

[service/networking/ingress-wildcard-host.yaml](#)



```
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: ingress-wildcard-host
spec:
  rules:
    - host: "foo.bar.com"
      http:
        paths:
          - pathType: Prefix
            path: "/bar"
            backend:
              service:
                name: servicel
                port:
                  number: 80
    - host: "*.*.foo.com"
      http:
        paths:
          - pathType: Prefix
            path: "/foo"
            backend:
              service:
                name: service2
                port:
                  number: 80
```

Ingress class

Ingresses can be implemented by different controllers, often with different configuration. Each Ingress should specify a class, a reference to an `IngressClass` resource that contains additional configuration including the name of the controller that should implement the class.

[service/networking/external-lb.yaml](#)



```
apiVersion: networking.k8s.io/v1
kind: IngressClass
metadata:
  name: external-lb
spec:
  controller: example.com/ingress-controller
  parameters:
    apiGroup: k8s.example.com
    kind: IngressParameters
    name: external-lb
```

IngressClass resources contain an optional parameters field. This can be used to reference additional configuration for this class.

Deprecated annotation

Before the IngressClass resource and ingressClassName field were added in Kubernetes 1.18, Ingress classes were specified with a kubernetes.io/ingress.class annotation on the Ingress. This annotation was never formally defined, but was widely supported by Ingress controllers.

The newer ingressClassName field on Ingresses is a replacement for that annotation, but is not a direct equivalent. While the annotation was generally used to reference the name of the Ingress controller that should implement the Ingress, the field is a reference to an IngressClass resource that contains additional Ingress configuration, including the name of the Ingress controller.

Default IngressClass

You can mark a particular IngressClass as default for your cluster. Setting the ingressclass.kubernetes.io/is-default-class annotation to true on an IngressClass resource will ensure that new Ingresses without an ingressClassName field specified will be assigned this default IngressClass.

Caution: If you have more than one IngressClass marked as the default for your cluster, the admission controller prevents creating new Ingress objects that don't have an ingressClassName specified. You can resolve this by ensuring that at most 1 IngressClass is marked as default in your cluster.

Types of Ingress

Ingress backed by a single Service

There are existing Kubernetes concepts that allow you to expose a single Service (see [alternatives](#)). You can also do this with an Ingress by specifying a default backend with no rules.

[service/networking/test-ingress.yaml](#)



```
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: test-ingress
spec:
  defaultBackend:
    service:
      name: test
      port:
        number: 80
```

If you create it using `kubectl apply -f` you should be able to view the state of the Ingress you just added:

```
kubectl get ingress test-ingress
```

NAME	CLASS	HOSTS	ADDRESS	PORTS	AGE
test-ingress	external-lb	*	203.0.113.123	80	59s

Where `203.0.113.123` is the IP allocated by the Ingress controller to satisfy this Ingress.

Note: Ingress controllers and load balancers may take a minute or two to allocate an IP address. Until that time, you often see the address listed as `<pending>`.

Simple fanout

A fanout configuration routes traffic from a single IP address to more than one Service, based on the HTTP URI being requested. An Ingress allows you to keep the number of load balancers down to a minimum. For example, a setup like:

[JavaScript must be [enabled](#) to view content]

would require an Ingress such as:

[service/networking/simple-fanout-example.yaml](#)

```
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: simple-fanout-example
spec:
  rules:
    - host: foo.bar.com
      http:
        paths:
          - path: /foo
            pathType: Prefix
            backend:
              service:
                name: service1
                port:
                  number: 4200
          - path: /bar
            pathType: Prefix
            backend:
              service:
                name: service2
                port:
                  number: 8080
```

When you create the Ingress with `kubectl apply -f`:

```
kubectl describe ingress simple-fanout-example
```

```
Name:           simple-fanout-example
Namespace:      default
Address:        178.91.123.132
Default backend: default-http-backend:80 (10.8.2.3:8080)
Rules:
  Host          Path  Backends
  ----          ---   -----
  foo.bar.com   /foo   service1:4200 (10.8.0.90:4200)
                 /bar   service2:8080 (10.8.0.91:8080)
Events:
  Type  Reason  Age   From
Message
  ----  -----  ---   ----
```

<i>Normal</i>	<i>ADD</i>	<i>22s</i>	<i>loadbalancer-controller</i>
<i>default/test</i>			

The Ingress controller provisions an implementation-specific load balancer that satisfies the Ingress, as long as the Services (`service1`, `service2`) exist. When it has done so, you can see the address of the load balancer at the Address field.

Note: Depending on the [Ingress controller](#) you are using, you may need to create a default-http-backend [Service](#).

Name based virtual hosting

Name-based virtual hosts support routing HTTP traffic to multiple host names at the same IP address.

[JavaScript must be [enabled](#) to view content]

The following Ingress tells the backing load balancer to route requests based on the [Host header](#).

[service/networking/name-virtual-host-ingress.yaml](#)

```
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: name-virtual-host-ingress
spec:
  rules:
    - host: foo.bar.com
      http:
        paths:
          - pathType: Prefix
            path: "/"
            backend:
              service:
                name: service1
                port:
                  number: 80
    - host: bar.foo.com
      http:
        paths:
          - pathType: Prefix
            path: "/"
            backend:
```

```
service:  
  name: service2  
  port:  
    number: 80
```

If you create an Ingress resource without any hosts defined in the rules, then any web traffic to the IP address of your Ingress controller can be matched without a name based virtual host being required.

For example, the following Ingress routes traffic requested for `first.bar.com` to `service1`, `second.foo.com` to `service2`, and any traffic to the IP address without a hostname defined in request (that is, without a request header being presented) to `service3`.

[service/networking/name-virtual-host-ingress-no-third-host.yaml](#) 

```
apiVersion: networking.k8s.io/v1  
kind: Ingress  
metadata:  
  name: name-virtual-host-ingress-no-third-host  
spec:  
  rules:  
    - host: first.bar.com  
      http:  
        paths:  
          - pathType: Prefix  
            path: "/"  
            backend:  
              service:  
                name: service1  
                port:  
                  number: 80  
    - host: second.bar.com  
      http:  
        paths:  
          - pathType: Prefix  
            path: "/"  
            backend:  
              service:  
                name: service2  
                port:  
                  number: 80  
    - http:  
      paths:  
        - pathType: Prefix  
          path: "/"  
          backend:  
            service:  
              name: service3
```

```
port:  
  number: 80
```

TLS

You can secure an Ingress by specifying a [Secret](#) that contains a TLS private key and certificate. The Ingress resource only supports a single TLS port, 443, and assumes TLS termination at the ingress point (traffic to the Service and its Pods is in plaintext). If the TLS configuration section in an Ingress specifies different hosts, they are multiplexed on the same port according to the hostname specified through the SNI TLS extension (provided the Ingress controller supports SNI). The TLS secret must contain keys named `tls.crt` and `tls.key` that contain the certificate and private key to use for TLS. For example:

```
apiVersion: v1  
kind: Secret  
metadata:  
  name: testsecret-tls  
  namespace: default  
data:  
  tls.crt: base64 encoded cert  
  tls.key: base64 encoded key  
type: kubernetes.io/tls
```

Referencing this secret in an Ingress tells the Ingress controller to secure the channel from the client to the load balancer using TLS. You need to make sure the TLS secret you created came from a certificate that contains a Common Name (CN), also known as a Fully Qualified Domain Name (FQDN) for `https-example.foo.com`.

Note: Keep in mind that TLS will not work on the default rule because the certificates would have to be issued for all the possible sub-domains. Therefore, hosts in the `tls` section need to explicitly match the host in the `rules` section.

[service/networking/tls-example-ingress.yaml](#)


```
apiVersion: networking.k8s.io/v1  
kind: Ingress  
metadata:  
  name: tls-example-ingress  
spec:  
  tls:  
    - hosts:  
        - https-example.foo.com
```

```
  secretName: testsecret-tls
  rules:
  - host: https-example.foo.com
    http:
      paths:
      - path: /
        pathType: Prefix
        backend:
          service:
            name: servicel
            port:
              number: 80
```

Note: There is a gap between TLS features supported by various Ingress controllers. Please refer to documentation on [nginx](#), [GCE](#), or any other platform specific Ingress controller to understand how TLS works in your environment.

Load balancing

An Ingress controller is bootstrapped with some load balancing policy settings that it applies to all Ingress, such as the load balancing algorithm, backend weight scheme, and others. More advanced load balancing concepts (e.g. persistent sessions, dynamic weights) are not yet exposed through the Ingress. You can instead get these features through the load balancer used for a Service.

It's also worth noting that even though health checks are not exposed directly through the Ingress, there exist parallel concepts in Kubernetes such as [readiness probes](#) that allow you to achieve the same end result. Please review the controller specific documentation to see how they handle health checks (for example: [nginx](#), or [GCE](#)).

Updating an Ingress

To update an existing Ingress to add a new Host, you can update it by editing the resource:

```
kubectl describe ingress test
```

```
Name:           test
Namespace:      default
Address:        178.91.123.132
Default backend: default-http-backend:80 (10.8.2.3:8080)
Rules:
```

<i>Host</i>	<i>Path</i>	<i>Backends</i>
<i>foo.bar.com</i>	<i>/foo</i>	<i>service1:80 (10.8.0.90:80)</i>
<i>Annotations:</i>		
<i>nginx.ingress.kubernetes.io/rewrite-target: /</i>		
<i>Events:</i>		
<i>Type</i>	<i>Reason</i>	<i>Age</i>
<i>Message</i>		
---	---	---
---	---	---
<i>Normal</i>	<i>ADD</i>	<i>35s</i>
<i>default/test</i>		
<i>From</i>		
<i>loadbalancer-controller</i>		

```
kubectl edit ingress test
```

This pops up an editor with the existing configuration in YAML format. Modify it to include the new Host:

```
spec:
  rules:
  - host: foo.bar.com
    http:
      paths:
        - backend:
            service:
              name: service1
              port:
                number: 80
            path: /foo
            pathType: Prefix
  - host: bar.baz.com
    http:
      paths:
        - backend:
            service:
              name: service2
              port:
                number: 80
            path: /foo
            pathType: Prefix
  ..
```

After you save your changes, `kubectl` updates the resource in the API server, which tells the Ingress controller to reconfigure the load balancer.

Verify this:

```
kubectl describe ingress test
```

```
Name:           test
Namespace:      default
Address:        178.91.123.132
Default backend: default-http-backend:80 (10.8.2.3:8080)
Rules:
Host          Path  Backends
----          ----  -----
foo.bar.com   /foo   service1:80 (10.8.0.90:80)
bar.baz.com   /foo   service2:80 (10.8.0.91:80)
Annotations:
  nginx.ingress.kubernetes.io/rewrite-target: /
Events:
Type    Reason  Age            From
Message
----    -----  --            --
-----  
Normal  ADD     45s           loadbalancer-controller
default/test
```

You can achieve the same outcome by invoking `kubectl replace -f` on a modified Ingress YAML file.

Failing across availability zones

Techniques for spreading traffic across failure domains differ between cloud providers. Please check the documentation of the relevant [Ingress controller](#) for details.

Alternatives

You can expose a Service in multiple ways that don't directly involve the Ingress resource:

- Use [Service.Type=LoadBalancer](#)
- Use [Service.Type=NodePort](#)

What's next

- Learn about the [Ingress API](#)
- Learn about [Ingress controllers](#)
- [Set up Ingress on Minikube with the NGINX Controller](#)

Feedback

Was this page helpful?

Yes No

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

Last modified August 28, 2020 at 3:31 PM PST: [Updated ingress.md TLS section \(73780ca1b\)](#)

[Edit this page](#) [Create child page](#) [Create an issue](#)

- [Terminology](#)
- [What is Ingress?](#)
- [Prerequisites](#)
- [The Ingress resource](#)
 - [Ingress rules](#)
 - [DefaultBackend](#)
 - [Resource backends](#)
 - [Path types](#)
 - [Examples](#)
- [Hostname wildcards](#)
- [Ingress class](#)
 - [Deprecated annotation](#)
 - [Default IngressClass](#)
- [Types of Ingress](#)
 - [Ingress backed by a single Service](#)
 - [Simple fanout](#)
 - [Name based virtual hosting](#)
 - [TLS](#)
 - [Load balancing](#)
- [Updating an Ingress](#)
- [Failing across availability zones](#)
- [Alternatives](#)
- [What's next](#)

Ingress Controllers

In order for the Ingress resource to work, the cluster must have an ingress controller running.

Unlike other types of controllers which run as part of the `kube-controller-manager` binary, Ingress controllers are not started automatically with a cluster. Use this page to choose the ingress controller implementation that best fits your cluster.

Kubernetes as a project supports and maintains [AWS](#), [GCE](#), and [nginx](#) ingress controllers.

Additional controllers

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

- [AKS Application Gateway Ingress Controller](#) is an ingress controller that configures the [Azure Application Gateway](#).
- [Ambassador API Gateway](#) is an [Envoy](#)-based ingress controller.
- The [Citrix ingress controller](#) works with Citrix Application Delivery Controller.
- [Contour](#) is an [Envoy](#) based ingress controller.
- F5 BIG-IP [Container Ingress Services for Kubernetes](#) lets you use an Ingress to configure F5 BIG-IP virtual servers.
- [Gloo](#) is an open-source ingress controller based on [Envoy](#), which offers API gateway functionality.
- [HAProxy Ingress](#) is an ingress controller for [HAProxy](#).
- The [HAProxy Ingress Controller for Kubernetes](#) is also an ingress controller for [HAProxy](#).
- [Istio Ingress](#) is an [Istio](#) based ingress controller.
- The [Kong Ingress Controller for Kubernetes](#) is an ingress controller driving [Kong Gateway](#).
- The [NGINX Ingress Controller for Kubernetes](#) works with the [NGINX](#) webserver (as a proxy).
- [Skipper](#) HTTP router and reverse proxy for service composition, including use cases like Kubernetes Ingress, designed as a library to build your custom proxy.
- The [Traefik Kubernetes Ingress provider](#) is an ingress controller for the [Traefik](#) proxy.
- [Voyager](#) is an ingress controller for [HAProxy](#).

Using multiple Ingress controllers

You may deploy [any number of ingress controllers](#) within a cluster. When you create an ingress, you should annotate each ingress with the appropriate [`ingress.class`](#) to indicate which ingress controller should be used if more than one exists within your cluster.

If you do not define a class, your cloud provider may use a default ingress controller.

Ideally, all ingress controllers should fulfill this specification, but the various ingress controllers operate slightly differently.

Note: Make sure you review your ingress controller's documentation to understand the caveats of choosing it.

What's next

- Learn more about [Ingress](#).
- [Set up Ingress on Minikube with the NGINX Controller](#).

Feedback

Was this page helpful?

Yes No

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

Last modified November 13, 2020 at 6:36 PM PST: [Revise ingress controller list \(53739e799\)](#)

[Edit this page](#) [Create child page](#) [Create an issue](#)

- [Additional controllers](#)
- [Using multiple Ingress controllers](#)
- [What's next](#)

Network Policies

If you want to control traffic flow at the IP address or port level (OSI layer 3 or 4), then you might consider using Kubernetes NetworkPolicies for particular applications in your cluster. NetworkPolicies are an application-centric construct which allow you to specify how a [pod](#) is allowed to communicate with various network "entities" (we use the word "entity" here to avoid overloading the more common terms such as "endpoints" and "services", which have specific Kubernetes connotations) over the network.

The entities that a Pod can communicate with are identified through a combination of the following 3 identifiers:

1. Other pods that are allowed (exception: a pod cannot block access to itself)
2. Namespaces that are allowed
3. IP blocks (exception: traffic to and from the node where a Pod is running is always allowed, regardless of the IP address of the Pod or the node)

When defining a pod- or namespace- based NetworkPolicy, you use a [selector](#) to specify what traffic is allowed to and from the Pod(s) that match the selector.

Meanwhile, when IP based NetworkPolicies are created, we define policies based on IP blocks (CIDR ranges).

Prerequisites

Network policies are implemented by the [network plugin](#). To use network policies, you must be using a networking solution which supports NetworkPolicy. Creating a NetworkPolicy resource without a controller that implements it will have no effect.

Isolated and Non-isolated Pods

By default, pods are non-isolated; they accept traffic from any source.

Pods become isolated by having a NetworkPolicy that selects them. Once there is any NetworkPolicy in a namespace selecting a particular pod, that pod will reject any connections that are not allowed by any NetworkPolicy. (Other pods in the namespace that are not selected by any NetworkPolicy will continue to accept all traffic.)

Network policies do not conflict; they are additive. If any policy or policies select a pod, the pod is restricted to what is allowed by the union of those policies' ingress/egress rules. Thus, order of evaluation does not affect the policy result.

The NetworkPolicy resource

See the [NetworkPolicy](#) reference for a full definition of the resource.

An example NetworkPolicy might look like this:

```
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: test-network-policy
  namespace: default
spec:
```

```

podSelector:
  matchLabels:
    role: db
policyTypes:
- Ingress
- Egress
ingress:
- from:
  - ipBlock:
    cidr: 172.17.0.0/16
    except:
    - 172.17.1.0/24
  - namespaceSelector:
    matchLabels:
      project: myproject
- podSelector:
    matchLabels:
      role: frontend
ports:
- protocol: TCP
  port: 6379
egress:
- to:
  - ipBlock:
    cidr: 10.0.0.0/24
ports:
- protocol: TCP
  port: 5978

```

Note: POSTing this to the API server for your cluster will have no effect unless your chosen networking solution supports network policy.

Mandatory Fields: As with all other Kubernetes config, a `NetworkPolicy` needs `apiVersion`, `kind`, and `metadata` fields. For general information about working with config files, see [Configure Containers Using a ConfigMap](#), and [Object Management](#).

spec: `NetworkPolicy spec` has all the information needed to define a particular network policy in the given namespace.

podSelector: Each `NetworkPolicy` includes a `podSelector` which selects the grouping of pods to which the policy applies. The example policy selects pods with the label "role=db". An empty `podSelector` selects all pods in the namespace.

policyTypes: Each `NetworkPolicy` includes a `policyTypes` list which may include either `Ingress`, `Egress`, or both. The `policyTypes` field indicates whether or not the given policy applies to ingress traffic to selected pod, egress traffic from selected pods, or both. If no `policyTypes` are specified on a `NetworkPolicy` then by default `Ingress` will always be set and `Egress` will be set if the `NetworkPolicy` has any egress rules.

ingress: Each NetworkPolicy may include a list of allowed ingress rules. Each rule allows traffic which matches both the from and ports sections. The example policy contains a single rule, which matches traffic on a single port, from one of three sources, the first specified via an ipBlock, the second via a namespaceSelector and the third via a podSelector.

egress: Each NetworkPolicy may include a list of allowed egress rules. Each rule allows traffic which matches both the to and ports sections. The example policy contains a single rule, which matches traffic on a single port to any destination in 10.0.0.0/24.

So, the example NetworkPolicy:

1. isolates "role=db" pods in the "default" namespace for both ingress and egress traffic (if they weren't already isolated)
2. (Ingress rules) allows connections to all pods in the "default" namespace with the label "role=db" on TCP port 6379 from:
 - any pod in the "default" namespace with the label "role=frontend"
 - any pod in a namespace with the label "project=myproject"
 - IP addresses in the ranges 172.17.0.0-172.17.0.255 and 172.17.2.0-172.17.255.255 (ie, all of 172.17.0.0/16 except 172.17.1.0/24)
3. (Egress rules) allows connections from any pod in the "default" namespace with the label "role=db" to CIDR 10.0.0.0/24 on TCP port 5978

See the [Declare Network Policy](#) walkthrough for further examples.

Behavior of to and from selectors

There are four kinds of selectors that can be specified in an ingress from section or egress to section:

podSelector: This selects particular Pods in the same namespace as the NetworkPolicy which should be allowed as ingress sources or egress destinations.

namespaceSelector: This selects particular namespaces for which all Pods should be allowed as ingress sources or egress destinations.

namespaceSelector and podSelector: A single to/from entry that specifies both namespaceSelector and podSelector selects particular Pods within particular namespaces. Be careful to use correct YAML syntax; this policy:

```
...
ingress:
- from:
  - namespaceSelector:
    matchLabels:
      user: alice
```

```
podSelector:  
  matchLabels:  
    role: client  
...
```

contains a single `from` element allowing connections from Pods with the label `role=client` in namespaces with the label `user=alice`. But this policy:

```
...  
ingress:  
  - from:  
    - namespaceSelector:  
        matchLabels:  
          user: alice  
    - podSelector:  
        matchLabels:  
          role: client  
...
```

contains two elements in the `from` array, and allows connections from Pods in the local Namespace with the label `role=client`, or from any Pod in any namespace with the label `user=alice`.

When in doubt, use `kubectl describe` to see how Kubernetes has interpreted the policy.

ipBlock: This selects particular IP CIDR ranges to allow as ingress sources or egress destinations. These should be cluster-external IPs, since Pod IPs are ephemeral and unpredictable.

Cluster ingress and egress mechanisms often require rewriting the source or destination IP of packets. In cases where this happens, it is not defined whether this happens before or after NetworkPolicy processing, and the behavior may be different for different combinations of network plugin, cloud provider, Service implementation, etc.

In the case of ingress, this means that in some cases you may be able to filter incoming packets based on the actual original source IP, while in other cases, the "source IP" that the NetworkPolicy acts on may be the IP of a LoadBalancer or of the Pod's node, etc.

For egress, this means that connections from pods to Service IPs that get rewritten to cluster-external IPs may or may not be subject to ipBlock-based policies.

Default policies

By default, if no policies exist in a namespace, then all ingress and egress traffic is allowed to and from pods in that namespace. The following examples let you change the default behavior in that namespace.

Default deny all ingress traffic

You can create a "default" isolation policy for a namespace by creating a `NetworkPolicy` that selects all pods but does not allow any ingress traffic to those pods.

[service/networking/network-policy-default-deny-ingress.yaml](#)



```
---  
apiVersion: networking.k8s.io/v1  
kind: NetworkPolicy  
metadata:  
  name: default-deny-ingress  
spec:  
  podSelector: {}  
  policyTypes:  
    - Ingress
```

This ensures that even pods that aren't selected by any other `NetworkPolicy` will still be isolated. This policy does not change the default egress isolation behavior.

Default allow all ingress traffic

If you want to allow all traffic to all pods in a namespace (even if policies are added that cause some pods to be treated as "isolated"), you can create a policy that explicitly allows all traffic in that namespace.

[service/networking/network-policy-allow-all-ingress.yaml](#)



```
---  
apiVersion: networking.k8s.io/v1  
kind: NetworkPolicy  
metadata:  
  name: allow-all-ingress  
spec:  
  podSelector: {}  
  ingress:  
    - {}  
  policyTypes:  
    - Ingress
```

Default deny all egress traffic

You can create a "default" egress isolation policy for a namespace by creating a `NetworkPolicy` that selects all pods but does not allow any egress traffic from those pods.

[service/networking/network-policy-default-deny-egress.yaml](#)



```
---  
apiVersion: networking.k8s.io/v1  
kind: NetworkPolicy  
metadata:  
  name: default-deny-egress  
spec:  
  podSelector: {}  
  policyTypes:  
    - Egress
```

This ensures that even pods that aren't selected by any other NetworkPolicy will not be allowed egress traffic. This policy does not change the default ingress isolation behavior.

Default allow all egress traffic

If you want to allow all traffic from all pods in a namespace (even if policies are added that cause some pods to be treated as "isolated"), you can create a policy that explicitly allows all egress traffic in that namespace.

[service/networking/network-policy-allow-all-egress.yaml](#)



```
---  
apiVersion: networking.k8s.io/v1  
kind: NetworkPolicy  
metadata:  
  name: allow-all-egress  
spec:  
  podSelector: {}  
  egress:  
    - {}  
  policyTypes:  
    - Egress
```

Default deny all ingress and all egress traffic

You can create a "default" policy for a namespace which prevents all ingress AND egress traffic by creating the following NetworkPolicy in that namespace.

[service/networking/network-policy-default-deny-all.yaml](#)



```
---  
apiVersion: networking.k8s.io/v1  
kind: NetworkPolicy  
metadata:
```

```
  name: default-deny-all
  spec:
    podSelector: {}
    policyTypes:
      - Ingress
      - Egress
```

This ensures that even pods that aren't selected by any other NetworkPolicy will not be allowed ingress or egress traffic.

SCTP support

FEATURE STATE: Kubernetes v1.19 [beta]

As a beta feature, this is enabled by default. To disable SCTP at a cluster level, you (or your cluster administrator) will need to disable the `SCTPSupport` [feature gate](#) for the API server with `--feature-gates=SCTPSupport=false`. When the feature gate is enabled, you can set the `protocol` field of a NetworkPolicy to `SCTP`.

Note: You must be using a [CNI](#) plugin that supports SCTP protocol NetworkPolicies.

What you can't do with network policies (at least, not yet)

As of Kubernetes 1.20, the following functionality does not exist in the NetworkPolicy API, but you might be able to implement workarounds using Operating System components (such as SELinux, OpenVSwitch, IPTables, and so on) or Layer 7 technologies (Ingress controllers, Service Mesh implementations) or admission controllers. In case you are new to network security in Kubernetes, its worth noting that the following User Stories cannot (yet) be implemented using the NetworkPolicy API. Some (but not all) of these user stories are actively being discussed for future releases of the NetworkPolicy API.

- Forcing internal cluster traffic to go through a common gateway (this might be best served with a service mesh or other proxy).
- Anything TLS related (use a service mesh or ingress controller for this).
- Node specific policies (you can use CIDR notation for these, but you cannot target nodes by their Kubernetes identities specifically).
- Targeting of namespaces or services by name (you can, however, target pods or namespaces by their [labels](#), which is often a viable workaround).
- Creation or management of "Policy requests" that are fulfilled by a third party.
- Default policies which are applied to all namespaces or pods (there are some third party Kubernetes distributions and projects which can do this).
- Advanced policy querying and reachability tooling.

- The ability to target ranges of Ports in a single policy declaration.
- The ability to log network security events (for example connections that are blocked or accepted).
- The ability to explicitly deny policies (currently the model for NetworkPolicies are deny by default, with only the ability to add allow rules).
- The ability to prevent loopback or incoming host traffic (Pods cannot currently block localhost access, nor do they have the ability to block access from their resident node).

What's next

- See the [Declare Network Policy](#) walkthrough for further examples.
- See more [recipes](#) for common scenarios enabled by the NetworkPolicy resource.

Feedback

Was this page helpful?

Yes No

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

Last modified November 03, 2020 at 10:54 AM PST: [fix heading level \(08b566111\)](#)

[Edit this page](#) [Create child page](#) [Create an issue](#)

- [Prerequisites](#)
- [Isolated and Non-isolated Pods](#)
- [The NetworkPolicy resource](#)
- [Behavior of to and from selectors](#)
- [Default policies](#)
 - [Default deny all ingress traffic](#)
 - [Default allow all ingress traffic](#)
 - [Default deny all egress traffic](#)
 - [Default allow all egress traffic](#)
 - [Default deny all ingress and all egress traffic](#)
- [SCTP support](#)
- [What you can't do with network policies \(at least, not yet\)](#)
- [What's next](#)

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 local host 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
      command:
        - cat
      args:
        - "/etc/hosts"

```

You can start a Pod with that configuration by running:

```
kubectl apply -f https://k8s.io/examples/service/networking/
hostaliases-pod.yaml
```

```
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	IP	NODE	
hostaliases-pod	0/1	Completed	0
6s	10.200.0.5	worker0	

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 Docker from [modifying](#) the file after the containers have already been started.

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.

Feedback

Was this page helpful?

Yes No

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

Last modified June 25, 2020 at 1:37 PM PST: [Update kubectl run from docs where necessary \(00f502fa6\)](#)

[Edit this page](#) [Create child page](#) [Create an issue](#)

- [Default hosts file content](#)
- [Adding additional entries with hostAliases](#)
- [Why does the kubelet manage the hosts file?](#)

IPv4/IPv6 dual-stack

FEATURE STATE: Kubernetes v1.16 [alpha]

IPv4/IPv6 dual-stack enables the allocation of both IPv4 and IPv6 addresses to [Pods](#) and [Services](#).

If you enable IPv4/IPv6 dual-stack networking for your Kubernetes cluster, the cluster will support the simultaneous assignment of both IPv4 and IPv6 addresses.

Supported Features

Enabling IPv4/IPv6 dual-stack on your Kubernetes cluster provides the following features:

- Dual-stack Pod networking (a single IPv4 and IPv6 address assignment per Pod)
- IPv4 and IPv6 enabled Services
- Pod off-cluster egress routing (eg. the Internet) via both IPv4 and IPv6 interfaces

Prerequisites

The following prerequisites are needed in order to utilize IPv4/IPv6 dual-stack Kubernetes clusters:

- Kubernetes 1.20 or later
For information about using dual-stack services with earlier Kubernetes versions, refer to the documentation for that version of Kubernetes.
- Provider support for dual-stack networking (Cloud provider or otherwise must be able to provide Kubernetes nodes with routable IPv4/IPv6 network interfaces)
- A network plugin that supports dual-stack (such as KubeNet or Calico)

Enable IPv4/IPv6 dual-stack

To enable IPv4/IPv6 dual-stack, enable the `IPv6DualStack` [feature gate](#) for the relevant components of your cluster, and set dual-stack cluster network assignments:

- `kube-apiserver`:
 - `--feature-gates="IPv6DualStack=true"`
 - `--service-cluster-ip-range=<IPv4 CIDR>, <IPv6 CIDR>`
- `kube-controller-manager`:
 - `--feature-gates="IPv6DualStack=true"`
 - `--cluster-cidr=<IPv4 CIDR>, <IPv6 CIDR>`
 - `--service-cluster-ip-range=<IPv4 CIDR>, <IPv6 CIDR>`
 - `--node-cidr-mask-size-ipv4/-node-cidr-mask-size-ipv6`
defaults to /24 for IPv4 and /64 for IPv6
- `kubelet`:
 - `--feature-gates="IPv6DualStack=true"`
- `kube-proxy`:
 - `--cluster-cidr=<IPv4 CIDR>, <IPv6 CIDR>`
 - `--feature-gates="IPv6DualStack=true"`

Note:

An example of an IPv4 CIDR: `10.244.0.0/16` (though you would supply your own address range)

An example of an IPv6 CIDR: `fdXY:IJKL:MNOP:15::/64` (this shows the format but is not a valid address - see [RFC 4193](#))

Services

If your cluster has dual-stack enabled, you can create [Services](#) which can use IPv4, IPv6, or both.

The address family of a Service defaults to the address family of the first service cluster IP range (configured via the `--service-cluster-ip-range` flag to the kube-controller-manager).

When you define a Service you can optionally configure it as dual stack. To specify the behavior you want, you set the `.spec.ipFamilyPolicy` field to one of the following values:

- `SingleStack`: Single-stack service. The control plane allocates a cluster IP for the Service, using the first configured service cluster IP range.
- `PreferDualStack`:
 - Only used if the cluster has dual-stack enabled. Allocates IPv4 and IPv6 cluster IPs for the Service
 - If the cluster does not have dual-stack enabled, this setting follows the same behavior as `SingleStack`.
- `RequireDualStack`: Allocates Service `.spec.ClusterIPs` from both IPv4 and IPv6 address ranges.
 - Selects the `.spec.ClusterIP` from the list of `.spec.ClusterIPs` based on the address family of the first element in the `.spec.ipFamilies` array.
 - The cluster must have dual-stack networking configured.

If you would like to define which IP family to use for single stack or define the order of IP families for dual-stack, you can choose the address families by setting an optional field, `.spec.ipFamilies`, on the Service.

Note: The `.spec.ipFamilies` field is immutable because the `.spec.ClusterIP` cannot be reallocated on a Service that already exists. If you want to change `.spec.ipFamilies`, delete and recreate the Service.

You can set `.spec.ipFamilies` to any of the following array values:

- `["IPv4"]`
- `["IPv6"]`
- `["IPv4", "IPv6"]` (dual stack)
- `["IPv6", "IPv4"]` (dual stack)

The first family you list is used for the legacy `.spec.ClusterIP` field.

Dual-stack Service configuration scenarios

These examples demonstrate the behavior of various dual-stack Service configuration scenarios.

Dual-stack options on new Services

1. This Service specification does not explicitly define `.spec.ipFamilyPolicy`. When you create this Service, Kubernetes assigns a cluster IP for the Service from the first configured `service-cluster-ip-range` and sets the `.spec.ipFamilyPolicy` to `SingleStack`. ([Services without selectors](#) and [headless Services](#) with selectors will behave in this same way.)

[`service/networking/dual-stack-default-svc.yaml`](#)


```
apiVersion: v1
kind: Service
metadata:
  name: my-service
  labels:
    app: MyApp
spec:
  selector:
    app: MyApp
  ports:
    - protocol: TCP
      port: 80
```

1. This Service specification explicitly defines `PreferDualStack` in `.spec.ipFamilyPolicy`. When you create this Service on a dual-stack cluster, Kubernetes assigns both IPv4 and IPv6 addresses for the service. The control plane updates the `.spec` for the Service to record the IP address assignments. The field `.spec.ClusterIPs` is the primary field, and contains both assigned IP addresses; `.spec.ClusterIP` is a secondary field with its value calculated from `.spec.ClusterIPs`.

- For the `.spec.ClusterIP` field, the control plane records the IP address that is from the same address family as the first service cluster IP range.
- On a single-stack cluster, the `.spec.ClusterIPs` and `.spec.ClusterIP` fields both only list one address.
- On a cluster with dual-stack enabled, specifying `RequireDualStack` in `.spec.ipFamilyPolicy` behaves the same as `PreferDualStack`.

[`service/networking/dual-stack-preferred-svc.yaml`](#)


```

apiVersion: v1
kind: Service
metadata:
  name: my-service
  labels:
    app: MyApp
spec:
  ipFamilyPolicy: PreferDualStack
  selector:
    app: MyApp
  ports:
    - protocol: TCP
      port: 80

```

1. This Service specification explicitly defines IPv6 and IPv4 in `.spec.ipFamilies` as well as defining `PreferDualStack` in `.spec.ipFamilyPolicy`. When Kubernetes assigns an IPv6 and IPv4 address in `.spec.ClusterIPs`, `.spec.ClusterIP` is set to the IPv6 address because that is the first element in the `.spec.ClusterIPs` array, overriding the default.

[service/networking/dual-stack-preferred-ipfamilies-svc.yaml](#)



```

apiVersion: v1
kind: Service
metadata:
  name: my-service
  labels:
    app: MyApp
spec:
  ipFamilyPolicy: PreferDualStack
  ipFamilies:
    - IPv6
    - IPv4
  selector:
    app: MyApp
  ports:
    - protocol: TCP
      port: 80

```

Dual-stack defaults on existing Services

These examples demonstrate the default behavior when dual-stack is newly enabled on a cluster where Services already exist.

1. When dual-stack is enabled on a cluster, existing Services (whether IPv4 or IPv6) are configured by the control plane to set `.spec.ipFamilyPolicy` to `SingleStack` and set `.spec.ipFamilies` to the address family of the existing Service. The existing Service cluster IP will be stored in `.spec.ClusterIPs`.

[service/networking/dual-stack-default-svc.yaml](#)



```
apiVersion: v1
kind: Service
metadata:
  name: my-service
  labels:
    app: MyApp
spec:
  selector:
    app: MyApp
  ports:
    - protocol: TCP
      port: 80
```

You can validate this behavior by using `kubectl` to inspect an existing service.

```
kubectl get svc my-service -o yaml
```

```
apiVersion: v1
kind: Service
metadata:
  labels:
    app: MyApp
  name: my-service
spec:
  clusterIP: 10.0.197.123
  clusterIPs:
  - 10.0.197.123
  ipFamilies:
  - IPv4
  ipFamilyPolicy: SingleStack
  ports:
  - port: 80
    protocol: TCP
    targetPort: 80
  selector:
    app: MyApp
  type: ClusterIP
status:
  loadBalancer: {}
```

1. When dual-stack is enabled on a cluster, existing [headless Services](#) with selectors are configured by the control plane to set `.spec.ipFamilyPolicy` to `SingleStack` and set `.spec.ipFamilies` to the address family of the first service cluster IP range (configured via the `--service-cluster-ip-range` flag to the `kube-controller-manager`) even though `.spec.ClusterIP` is set to `None`.

[service/networking/dual-stack-default-svc.yaml](#)



```
apiVersion: v1
kind: Service
metadata:
  name: my-service
  labels:
    app: MyApp
spec:
  selector:
    app: MyApp
  ports:
    - protocol: TCP
      port: 80
```

You can validate this behavior by using `kubectl` to inspect an existing headless service with selectors.

```
kubectl get svc my-service -o yaml
```

```
apiVersion: v1
kind: Service
metadata:
  labels:
    app: MyApp
  name: my-service
spec:
  clusterIP: None
  clusterIPs:
  - None
  ipFamilies:
  - IPv4
  ipFamilyPolicy: SingleStack
  ports:
  - port: 80
    protocol: TCP
    targetPort: 80
  selector:
    app: MyApp
```

Switching Services between single-stack and dual-stack

Services can be changed from single-stack to dual-stack and from dual-stack to single-stack.

1. To change a Service from single-stack to dual-stack, change `.spec.ipFamilyPolicy` from `SingleStack` to `PreferDualStack` or `RequireDualStack` as desired. When you change this Service from single-stack to dual-stack, Kubernetes assigns the missing address family so that the Service now has IPv4 and IPv6 addresses.

Edit the Service specification updating the `.spec.ipFamilyPolicy` from `SingleStack` to `PreferDualStack`.

Before:

```
spec:  
  ipFamilyPolicy: SingleStack
```

After:

```
spec:  
  ipFamilyPolicy: PreferDualStack
```

1. To change a Service from dual-stack to single-stack, change `.spec.ipFamilyPolicy` from `PreferDualStack` or `RequireDualStack` to `SingleStack`. When you change this Service from dual-stack to single-stack, Kubernetes retains only the first element in the `.spec.ClusterIPs` array, and sets `.spec.ClusterIP` to that IP address and sets `.spec.ipFamilies` to the address family of `.spec.ClusterIPs`.

Headless Services without selector

For [Headless Services without selectors](#) and without `.spec.ipFamilyPolicy` explicitly set, the `.spec.ipFamilyPolicy` field defaults to `RequireDualStack`.

Service type LoadBalancer

To provision a dual-stack load balancer for your Service:

- Set the `.spec.type` field to `LoadBalancer`
- Set `.spec.ipFamilyPolicy` field to `PreferDualStack` or `RequireDualStack`

Note: To use a dual-stack `LoadBalancer` type Service, your cloud provider must support IPv4 and IPv6 load balancers.

Egress traffic

If you want to enable egress traffic in order to reach off-cluster destinations (eg. the public Internet) from a Pod that uses non-publicly routable IPv6 addresses, you need to enable the Pod to use a publicly routed IPv6 address via a mechanism such as transparent proxying or IP masquerading. The [ip-masq-agent](#) project supports IP masquerading on dual-stack clusters.

Note: Ensure your [CNI](#) provider supports IPv6.

What's next

- [Validate IPv4/IPv6 dual-stack networking](#)

Feedback

Was this page helpful?

Yes No

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

Last modified October 26, 2020 at 1:06 PM PST: [Dual-stack docs for Kubernetes 1.20 \(8a3244fdd\)](#)

[Edit this page](#) [Create child page](#) [Create an issue](#)

- [Supported Features](#)
- [Prerequisites](#)
- [Enable IPv4/IPv6 dual-stack](#)
- [Services](#)
 - [Dual-stack Service configuration scenarios](#)
 - [Headless Services without selector](#)
 - [Service type LoadBalancer](#)
- [Egress traffic](#)
- [What's next](#)

Storage

Ways to provide both long-term and temporary storage to Pods in your cluster.

[Volumes](#)

[Persistent Volumes](#)

[Volume Snapshots](#)

[CSI Volume Cloning](#)

[Storage Classes](#)

[Volume Snapshot Classes](#)

[Dynamic Volume Provisioning](#)

[Storage Capacity](#)

[Ephemeral Volumes](#)

[Node-specific Volume Limits](#)

Volumes

On-disk files in a container are ephemeral, which presents some problems for non-trivial applications when running in containers. One problem is the loss of files when a container crashes. The kubelet restarts the container but with a clean state. A second problem occurs when sharing files between containers running together in a Pod. The Kubernetes [volume](#) abstraction solves both of these problems. Familiarity with [Pods](#) is suggested.

Background

Docker has a concept of [volumes](#), though it is somewhat looser and less managed. A Docker volume is a directory on disk or in another container. Docker provides volume drivers, but the functionality is somewhat limited.

Kubernetes supports many types of volumes. A [Pod](#) can use any number of volume types simultaneously. Ephemeral volume types have a lifetime of a pod, but persistent volumes exist beyond the lifetime of a pod. Consequently, a volume outlives any containers that run within the pod, and data is preserved across container restarts. When a pod ceases to exist, the volume is destroyed.

At its core, a volume is just a directory, possibly with some data in it, which is accessible to the containers in a pod. How that directory comes to be, the medium that backs it, and the contents of it are determined by the particular volume type used.

To use a volume, specify the volumes to provide for the Pod in `.spec.volumes` and declare where to mount those volumes into containers in `.spec.containers[].volumeMounts`. A process in a container sees a filesystem view composed from their Docker image and volumes. The [Docker image](#) is at the root of the filesystem hierarchy. Volumes mount at the specified paths within the image. Volumes can not mount onto other volumes or have hard links to other volumes. Each Container in the Pod's configuration must independently specify where to mount each volume.*

Types of Volumes

Kubernetes supports several types of volumes.

awsElasticBlockStore

An `awsElasticBlockStore` volume mounts an Amazon Web Services (AWS) [EBS volume](#) into your pod. Unlike `emptyDir`, which is erased when a pod is removed, the contents of an EBS volume are persisted and the volume is unmounted. This means that an EBS volume can be pre-populated with data, and that data can be shared between pods.

Note: You must create an EBS volume by using `aws ec2 create-volume` or the AWS API before you can use it.

There are some restrictions when using an `awsElasticBlockStore` volume:

- the nodes on which pods are running must be AWS EC2 instances
- those instances need to be in the same region and availability zone as the EBS volume
- EBS only supports a single EC2 instance mounting a volume

Creating an AWS EBS volume

Before you can use an EBS volume with a pod, you need to create it.

```
aws ec2 create-volume --availability-zone=eu-west-1a --size=10 --volume-type=gp2
```

Make sure the zone matches the zone you brought up your cluster in. Check that the size and EBS volume type are suitable for your use.

AWS EBS configuration example

```
apiVersion: v1
kind: Pod
metadata:
  name: test-ebs
spec:
  containers:
    - image: k8s.gcr.io/test-webserver
      name: test-container
      volumeMounts:
        - mountPath: /test-ebs
          name: test-volume
  volumes:
    - name: test-volume
      # This AWS EBS volume must already exist.
      awsElasticBlockStore:
        volumeID: "<volume id>"
        fsType: ext4
```

AWS EBS CSI migration

FEATURE STATE: Kubernetes v1.17 [beta]

The `CSIMigration` feature for `awsElasticBlockStore`, when enabled, redirects all plugin operations from the existing in-tree plugin to the `ebs.csi.aws.com` Container Storage Interface (CSI) driver. In order to use this feature, the [AWS EBS CSI driver](#) must be installed on the cluster and the `CSIMigration` and `CSIMigrationAWS` beta features must be enabled.

AWS EBS CSI migration complete

FEATURE STATE: Kubernetes v1.17 [alpha]

To disable the `awsElasticBlockStore` storage plugin from being loaded by the controller manager and the kubelet, set the `CSIMigrationAWSComplete` flag to `true`. This feature requires the `ebs.csi.aws.com` Container Storage Interface (CSI) driver installed on all worker nodes.

azureDisk

The `azureDisk` volume type mounts a Microsoft Azure [Data Disk](#) into a pod.

For more details, see the [azureDisk volume plugin](#).

azureDisk CSI migration

FEATURE STATE: Kubernetes v1.19 [beta]

The `CSIMigration` feature for `azureDisk`, when enabled, redirects all plugin operations from the existing in-tree plugin to the `disk.csi.azure.com` Container Storage Interface (CSI) Driver. In order to use this feature, the [Azure Disk CSI Driver](#) must be installed on the cluster and the `CSIMigration` and `CSIMigrationAzureDisk` features must be enabled.

azureFile

The `azureFile` volume type mounts a Microsoft Azure File volume (SMB 2.1 and 3.0) into a pod.

For more details, see the [azureFile volume plugin](#).

azureFile CSI migration

FEATURE STATE: Kubernetes v1.15 [alpha]

The `CSIMigration` feature for `azureFile`, when enabled, redirects all plugin operations from the existing in-tree plugin to the `file.csi.azure.com` Container Storage Interface (CSI) Driver. In order to use this feature, the [Azure File CSI Driver](#) must be installed on the cluster and the `CSIMigration` and `CSIMigrationAzureFile` alpha features must be enabled.

cephfs

A `cephfs` volume allows an existing CephFS volume to be mounted into your Pod. Unlike `emptyDir`, which is erased when a pod is removed, the contents of a `cephfs` volume are preserved and the volume is merely unmounted. This means that a `cephfs` volume can be pre-populated with data, and that data can be shared between pods. The `cephfs` volume can be mounted by multiple writers simultaneously.

Note: You must have your own Ceph server running with the share exported before you can use it.

See the [CephFS example](#) for more details.

cinder

Note: Kubernetes must be configured with the OpenStack cloud provider.

The *cinder* volume type is used to mount the OpenStack Cinder volume into your pod.

Cinder volume configuration example

```
apiVersion: v1
kind: Pod
metadata:
  name: test-cinder
spec:
  containers:
    - image: k8s.gcr.io/test-webserver
      name: test-cinder-container
      volumeMounts:
        - mountPath: /test-cinder
          name: test-volume
    volumes:
      - name: test-volume
        # This OpenStack volume must already exist.
        cinder:
          volumeID: "<volume id>"
          fsType: ext4
```

OpenStack CSI migration

FEATURE STATE: Kubernetes v1.18 [beta]

The *CSIMigration* feature for Cinder, when enabled, redirects all plugin operations from the existing in-tree plugin to the *cinder.csi.openstack.org* Container Storage Interface (CSI) Driver. In order to use this feature, the [Openstack Cinder CSI Driver](#) must be installed on the cluster and the *CSIMigration* and *CSIMigrationOpenStack* beta features must be enabled.

configMap

A [ConfigMap](#) provides a way to inject configuration data into pods. The data stored in a ConfigMap can be referenced in a volume of type *configMap* and then consumed by containerized applications running in a pod.

When referencing a ConfigMap, you provide the name of the ConfigMap in the volume. You can customize the path to use for a specific entry in the ConfigMap. The following configuration shows how to mount the *log-config* ConfigMap onto a Pod called *configmap-pod*:

```
apiVersion: v1
kind: Pod
```

```

metadata:
  name: configmap-pod
spec:
  containers:
    - name: test
      image: busybox
      volumeMounts:
        - name: config-vol
          mountPath: /etc/config
  volumes:
    - name: config-vol
      configMap:
        name: log-config
        items:
          - key: log_level
            path: log_level

```

The `log-config` ConfigMap is mounted as a volume, and all contents stored in its `log_level` entry are mounted into the Pod at path `/etc/config/log_level`. Note that this path is derived from the volume's `mountPath` and the path keyed with `log_level`.

Note:

- You must create a [ConfigMap](#) before you can use it.
- A container using a ConfigMap as a [subPath](#) volume mount will not receive ConfigMap updates.
- Text data is exposed as files using the UTF-8 character encoding. For other character encodings, use `binaryData`.

downwardAPI

A `downwardAPI` volume makes downward API data available to applications. It mounts a directory and writes the requested data in plain text files.

Note: A container using the downward API as a [subPath](#) volume mount will not receive downward API updates.

See the [downward API example](#) for more details.

emptyDir

An `emptyDir` volume is first created when a Pod is assigned to a node, and exists as long as that Pod is running on that node. As the name says, the `emptyDir` volume is initially empty. All containers in the Pod can read and write the same files in the `emptyDir` volume, though that volume can be mounted at the same or different paths in each container. When a Pod is removed from a node for any reason, the data in the `emptyDir` is deleted permanently.

Note: A container crashing does not remove a Pod from a node. The data in an `emptyDir` volume is safe across container crashes.

Some uses for an `emptyDir` are:

- scratch space, such as for a disk-based merge sort
- checkpointing a long computation for recovery from crashes
- holding files that a content-manager container fetches while a webserver container serves the data

Depending on your environment, `emptyDir` volumes are stored on whatever medium that backs the node such as disk or SSD, or network storage. However, if you set the `emptyDir.medium` field to "Memory", Kubernetes mounts a `tmpfs` (RAM-backed filesystem) for you instead. While `tmpfs` is very fast, be aware that unlike disks, `tmpfs` is cleared on node reboot and any files you write count against your container's memory limit.

Note: If the `SizeMemoryBackedVolumes` [feature gate](#) is enabled, you can specify a size for memory backed volumes. If no size is specified, memory backed volumes are sized to 50% of the memory on a Linux host.

`emptyDir` configuration example

```
apiVersion: v1
kind: Pod
metadata:
  name: test-pd
spec:
  containers:
    - image: k8s.gcr.io/test-webserver
      name: test-container
      volumeMounts:
        - mountPath: /cache
          name: cache-volume
  volumes:
    - name: cache-volume
      emptyDir: {}
```

`fc` (*fibre channel*)

An `fc` volume type allows an existing fibre channel block storage volume to mount in a Pod. You can specify single or multiple target world wide names (WWNs) using the parameter `targetWNNs` in your Volume configuration. If multiple WWNs are specified, `targetWNNs` expect that those WWNs are from multi-path connections.

Note: You must configure FC SAN Zoning to allocate and mask those LUNs (volumes) to the target WWNs beforehand so that Kubernetes hosts can access them.

See the [fibre channel example](#) for more details.

flocker (deprecated)

[Flocker](#) is an open-source, clustered container data volume manager. Flocker provides management and orchestration of data volumes backed by a variety of storage backends.

A *flocker* volume allows a Flocker dataset to be mounted into a Pod. If the dataset does not already exist in Flocker, it needs to be first created with the Flocker CLI or by using the Flocker API. If the dataset already exists it will be reattached by Flocker to the node that the pod is scheduled. This means data can be shared between pods as required.

Note: You must have your own Flocker installation running before you can use it.

See the [Flocker example](#) for more details.

gcePersistentDisk

A *gcePersistentDisk* volume mounts a Google Compute Engine (GCE) [persistent disk](#) (PD) into your Pod. Unlike `emptyDir`, which is erased when a pod is removed, the contents of a PD are preserved and the volume is merely unmounted. This means that a PD can be pre-populated with data, and that data can be shared between pods.

Note: You must create a PD using `gcloud` or the GCE API or UI before you can use it.

There are some restrictions when using a *gcePersistentDisk*:

- the nodes on which Pods are running must be GCE VMs
- those VMs need to be in the same GCE project and zone as the persistent disk

One feature of GCE persistent disk is concurrent read-only access to a persistent disk. A *gcePersistentDisk* volume permits multiple consumers to simultaneously mount a persistent disk as read-only. This means that you can pre-populate a PD with your dataset and then serve it in parallel from as many Pods as you need. Unfortunately, PDs can only be mounted by a single consumer in read-write mode. Simultaneous writers are not allowed.

Using a GCE persistent disk with a Pod controlled by a ReplicaSet will fail unless the PD is read-only or the replica count is 0 or 1.

Creating a GCE persistent disk

Before you can use a GCE persistent disk with a Pod, you need to create it.

```
gcloud compute disks create --size=500GB --zone=us-central1-a my-data-disk
```

GCE persistent disk configuration example

```
apiVersion: v1
kind: Pod
metadata:
  name: test-pd
spec:
  containers:
    - image: k8s.gcr.io/test-webserver
      name: test-container
      volumeMounts:
        - mountPath: /test-pd
          name: test-volume
  volumes:
    - name: test-volume
      # This GCE PD must already exist.
      gcePersistentDisk:
        pdName: my-data-disk
        fsType: ext4
```

Regional persistent disks

The [Regional persistent disks](#) feature allows the creation of persistent disks that are available in two zones within the same region. In order to use this feature, the volume must be provisioned as a PersistentVolume; referencing the volume directly from a pod is not supported.

Manually provisioning a Regional PD PersistentVolume

Dynamic provisioning is possible using a [StorageClass for GCE PD](#). Before creating a PersistentVolume, you must create the persistent disk:

```
gcloud compute disks create --size=500GB my-data-disk
  --region us-central1
  --replica-zones us-central1-a,us-central1-b
```

Regional persistent disk configuration example

```
apiVersion: v1
kind: PersistentVolume
metadata:
  name: test-volume
spec:
  capacity:
    storage: 400Gi
  accessModes:
    - ReadWriteOnce
  gcePersistentDisk:
    pdName: my-data-disk
    fsType: ext4
  nodeAffinity:
```

```

required:
  nodeSelectorTerms:
    - matchExpressions:
      - key: failure-domain.beta.kubernetes.io/zone
        operator: In
        values:
          - us-central1-a
          - us-central1-b

```

GCE CSI migration

FEATURE STATE: Kubernetes v1.17 [beta]

The *CSIMigration* feature for GCE PD, when enabled, redirects all plugin operations from the existing in-tree plugin to the `pd.csi.storage.gke.io` Container Storage Interface (CSI) Driver. In order to use this feature, the [GCE PD CSI Driver](#) must be installed on the cluster and the `CSIMigration` and `CSIMigrationGCE` beta features must be enabled.

gitRepo (deprecated)

Warning: The `gitRepo` volume type is deprecated. To provision a container with a git repo, mount an [EmptyDir](#) into an `InitContainer` that clones the repo using `git`, then mount the [EmptyDir](#) into the Pod's container.

A `gitRepo` volume is an example of a volume plugin. This plugin mounts an empty directory and clones a git repository into this directory for your Pod to use.

Here is an example of a `gitRepo` volume:

```

apiVersion: v1
kind: Pod
metadata:
  name: server
spec:
  containers:
    - image: nginx
      name: nginx
      volumeMounts:
        - mountPath: /mypath
          name: git-volume
  volumes:
    - name: git-volume
  gitRepo:
    repository: "git@somewhere:me/my-git-repository.git"
    revision: "22f1d8406d464b0c0874075539c1f2e96c253775"

```

glusterfs

A `glusterfs` volume allows a [Glusterfs](#) (an open source networked filesystem) volume to be mounted into your Pod. Unlike `emptyDir`, which is erased when a Pod is removed, the contents of a `glusterfs` volume are preserved and the volume is merely unmounted. This means that a `glusterfs` volume can be pre-populated with data, and that data can be shared between pods. GlusterFS can be mounted by multiple writers simultaneously.

Note: You must have your own GlusterFS installation running before you can use it.

See the [GlusterFS example](#) for more details.

hostPath

A `hostPath` volume mounts a file or directory from the host node's filesystem into your Pod. This is not something that most Pods will need, but it offers a powerful escape hatch for some applications.

For example, some uses for a `hostPath` are:

- running a container that needs access to Docker internals; use a `hostPath` of `/var/lib/docker`
- running cAdvisor in a container; use a `hostPath` of `/sys`
- allowing a Pod to specify whether a given `hostPath` should exist prior to the Pod running, whether it should be created, and what it should exist as

In addition to the required `path` property, you can optionally specify a `type` for a `hostPath` volume.

The supported values for field `type` are:

Value	Behavior
	Empty string (default) is for backward compatibility, which means that no checks will be performed before mounting the <code>hostPath</code> volume.
<code>DirectoryOrCreate</code>	If nothing exists at the given path, an empty directory will be created there as needed with permission set to 0755, having the same group and ownership with Kubelet.
<code>Directory</code>	A directory must exist at the given path
<code>FileOrCreate</code>	If nothing exists at the given path, an empty file will be created there as needed with permission set to 0644, having the same group and ownership with Kubelet.
<code>File</code>	A file must exist at the given path
<code>Socket</code>	A UNIX socket must exist at the given path
<code>CharDevice</code>	A character device must exist at the given path
<code>BlockDevice</code>	A block device must exist at the given path

Watch out when using this type of volume, because:

- *Pods with identical configuration (such as created from a PodTemplate) may behave differently on different nodes due to different files on the nodes*
- *The files or directories created on the underlying hosts are only writable by root. You either need to run your process as root in a [privileged Container](#) or modify the file permissions on the host to be able to write to a hostPath volume*

hostPath configuration example

```
apiVersion: v1
kind: Pod
metadata:
  name: test-pd
spec:
  containers:
    - image: k8s.gcr.io/test-webserver
      name: test-container
      volumeMounts:
        - mountPath: /test-pd
          name: test-volume
    volumes:
      - name: test-volume
        hostPath:
          # directory location on host
          path: /data
          # this field is optional
          type: Directory
```

Caution: The `FileOrCreate` mode does not create the parent directory of the file. If the parent directory of the mounted file does not exist, the pod fails to start. To ensure that this mode works, you can try to mount directories and files separately, as shown in the [FileOrCreateconfiguration](#).

hostPath FileOrCreate configuration example

```
apiVersion: v1
kind: Pod
metadata:
  name: test-webserver
spec:
  containers:
    - name: test-webserver
      image: k8s.gcr.io/test-webserver:latest
      volumeMounts:
        - mountPath: /var/local/aaa
          name: mydir
        - mountPath: /var/local/aaa/1.txt
```

```
  name: myfile
volumes:
- name: mydir
  hostPath:
    # Ensure the file directory is created.
    path: /var/local/aaa
    type: DirectoryOrCreate
- name: myfile
  hostPath:
    path: /var/local/aaa/1.txt
    type: FileOrCreate
```

iscsi

An *iscsi* volume allows an existing iSCSI (SCSI over IP) volume to be mounted into your Pod. Unlike *emptyDir*, which is erased when a Pod is removed, the contents of an *iscsi* volume are preserved and the volume is merely unmounted. This means that an *iscsi* volume can be pre-populated with data, and that data can be shared between pods.

Note: You must have your own iSCSI server running with the volume created before you can use it.

A feature of iSCSI is that it can be mounted as read-only by multiple consumers simultaneously. This means that you can pre-populate a volume with your dataset and then serve it in parallel from as many Pods as you need. Unfortunately, iSCSI volumes can only be mounted by a single consumer in read-write mode. Simultaneous writers are not allowed.

See the [iSCSI example](#) for more details.

local

A *local* volume represents a mounted local storage device such as a disk, partition or directory.

Local volumes can only be used as a statically created PersistentVolume. Dynamic provisioning is not supported.

Compared to *hostPath* volumes, *local* volumes are used in a durable and portable manner without manually scheduling pods to nodes. The system is aware of the volume's node constraints by looking at the node affinity on the PersistentVolume.

However, *local* volumes are subject to the availability of the underlying node and are not suitable for all applications. If a node becomes unhealthy, then the *local* volume becomes inaccessible by the pod. The pod using this volume is unable to run. Applications using *local* volumes must be able to tolerate this reduced availability, as well as potential data loss, depending on the durability characteristics of the underlying disk.

The following example shows a `PersistentVolume` using a `local` volume and `nodeAffinity`:

```
apiVersion: v1
kind: PersistentVolume
metadata:
  name: example-pv
spec:
  capacity:
    storage: 100Gi
  volumeMode: Filesystem
  accessModes:
    - ReadWriteOnce
  persistentVolumeReclaimPolicy: Delete
  storageClassName: local-storage
  local:
    path: /mnt/disks/ssd1
  nodeAffinity:
    required:
      nodeSelectorTerms:
        - matchExpressions:
            - key: kubernetes.io/hostname
              operator: In
            values:
              - example-node
```

You must set a `PersistentVolume` `nodeAffinity` when using `local` volumes. The Kubernetes scheduler uses the `PersistentVolume` `nodeAffinity` to schedule these Pods to the correct node.

`PersistentVolume` `volumeMode` can be set to "Block" (instead of the default value "Filesystem") to expose the local volume as a raw block device.

When using local volumes, it is recommended to create a `StorageClass` with `volumeBindingMode` set to `WaitForFirstConsumer`. For more details, see the local [StorageClass](#) example. Delaying volume binding ensures that the `PersistentVolumeClaim` binding decision will also be evaluated with any other node constraints the Pod may have, such as node resource requirements, node selectors, Pod affinity, and Pod anti-affinity.

An external static provisioner can be run separately for improved management of the local volume lifecycle. Note that this provisioner does not support dynamic provisioning yet. For an example on how to run an external local provisioner, see the [local volume provisioner user guide](#).

Note: The local `PersistentVolume` requires manual cleanup and deletion by the user if the external static provisioner is not used to manage the volume lifecycle.

nfs

An *nfs* volume allows an existing NFS (Network File System) share to be mounted into a Pod. Unlike *emptyDir*, which is erased when a Pod is removed, the contents of an *nfs* volume are preserved and the volume is merely unmounted. This means that an NFS volume can be pre-populated with data, and that data can be shared between pods. NFS can be mounted by multiple writers simultaneously.

Note: You must have your own NFS server running with the share exported before you can use it.

See the [NFS example](#) for more details.

persistentVolumeClaim

A *persistentVolumeClaim* volume is used to mount a [PersistentVolume](#) into a Pod. PersistentVolumeClaims are a way for users to "claim" durable storage (such as a GCE PersistentDisk or an iSCSI volume) without knowing the details of the particular cloud environment.

See the information about [PersistentVolumes](#) for more details.

portworxVolume

A *portworxVolume* is an elastic block storage layer that runs hyperconverged with Kubernetes. [Portworx](#) fingerprints storage in a server, tiers based on capabilities, and aggregates capacity across multiple servers. Portworx runs in-guest in virtual machines or on bare metal Linux nodes.

A *portworxVolume* can be dynamically created through Kubernetes or it can also be pre-provisioned and referenced inside a Pod. Here is an example Pod referencing a pre-provisioned Portworx volume:

```
apiVersion: v1
kind: Pod
metadata:
  name: test-portworx-volume-pod
spec:
  containers:
    - image: k8s.gcr.io/test-webserver
      name: test-container
      volumeMounts:
        - mountPath: /mnt
          name: pxvol
  volumes:
    - name: pxvol
      # This Portworx volume must already exist.
      portworxVolume:
        volumeID: "pxvol"
        fsType: "<fs-type>"
```

Note: Make sure you have an existing PortworxVolume with name `pxvol` before using it in the Pod.

For more details, see the [Portworx volume](#) examples.

projected

A projected volume maps several existing volume sources into the same directory.

Currently, the following types of volume sources can be projected:

- [secret](#)
- [downwardAPI](#)
- [configMap](#)
- `serviceAccountToken`

All sources are required to be in the same namespace as the Pod. For more details, see the [all-in-one volume design document](#).

Example configuration with a secret, a downwardAPI, and a configMap

```
apiVersion: v1
kind: Pod
metadata:
  name: volume-test
spec:
  containers:
    - name: container-test
      image: busybox
      volumeMounts:
        - name: all-in-one
          mountPath: "/projected-volume"
          readOnly: true
  volumes:
    - name: all-in-one
      projected:
        sources:
          - secret:
              name: mysecret
              items:
                - key: username
                  path: my-group/my-username
          - downwardAPI:
              items:
                - path: "labels"
                  fieldRef:
                    fieldPath: metadata.labels
                - path: "cpu_limit"
                  resourceFieldRef:
                    containerName: container-test
```

```

    resource: limits.cpu
- configMap:
  name: myconfigmap
  items:
  - key: config
    path: my-group/my-config

```

Example configuration: secrets with a non-default permission mode set

```

apiVersion: v1
kind: Pod
metadata:
  name: volume-test
spec:
  containers:
  - name: container-test
    image: busybox
    volumeMounts:
    - name: all-in-one
      mountPath: "/projected-volume"
      readOnly: true
  volumes:
  - name: all-in-one
    projected:
      sources:
      - secret:
          name: mysecret
          items:
          - key: username
            path: my-group/my-username
      - secret:
          name: mysecret2
          items:
          - key: password
            path: my-group/my-password
            mode: 511

```

Each projected volume source is listed in the spec under sources. The parameters are nearly the same with two exceptions:

- For secrets, the `secretName` field has been changed to `name` to be consistent with ConfigMap naming.
- The `defaultMode` can only be specified at the projected level and not for each volume source. However, as illustrated above, you can explicitly set the `mode` for each individual projection.

When the `TokenRequestProjection` feature is enabled, you can inject the token for the current [service account](#) into a Pod at a specified path. For example:

```
apiVersion: v1
kind: Pod
metadata:
  name: sa-token-test
spec:
  containers:
    - name: container-test
      image: busybox
      volumeMounts:
        - name: token-vol
          mountPath: "/service-account"
          readOnly: true
  volumes:
    - name: token-vol
      projected:
        sources:
          - serviceAccountToken:
              audience: api
              expirationSeconds: 3600
              path: token
```

The example Pod has a projected volume containing the injected service account token. This token can be used by a Pod's containers to access the Kubernetes API server. The audience field contains the intended audience of the token. A recipient of the token must identify itself with an identifier specified in the audience of the token, and otherwise should reject the token. This field is optional and it defaults to the identifier of the API server.

The expirationSeconds is the expected duration of validity of the service account token. It defaults to 1 hour and must be at least 10 minutes (600 seconds). An administrator can also limit its maximum value by specifying the --service-account-max-token-expiration option for the API server. The path field specifies a relative path to the mount point of the projected volume.

Note: A container using a projected volume source as a [subPath](#) volume mount will not receive updates for those volume sources.

quobyte

A quobyte volume allows an existing [Quobyte](#) volume to be mounted into your Pod.

Note: You must have your own Quobyte setup and running with the volumes created before you can use it.

Quobyte supports the [Container Storage Interface](#). CSI is the recommended plugin to use Quobyte volumes inside Kubernetes. Quobyte's GitHub project has [instructions](#) for deploying Quobyte using CSI, along with examples.

rbd

An `rbd` volume allows a [Rados Block Device \(RBD\)](#) volume to mount into your Pod. Unlike `emptyDir`, which is erased when a pod is removed, the contents of an `rbd` volume are preserved and the volume is unmounted. This means that a RBD volume can be pre-populated with data, and that data can be shared between pods.

Note: You must have a Ceph installation running before you can use RBD.

A feature of RBD is that it can be mounted as read-only by multiple consumers simultaneously. This means that you can pre-populate a volume with your dataset and then serve it in parallel from as many pods as you need. Unfortunately, RBD volumes can only be mounted by a single consumer in read-write mode. Simultaneous writers are not allowed.

See the [RBD example](#) for more details.

scaleIO (deprecated)

ScaleIO is a software-based storage platform that uses existing hardware to create clusters of scalable shared block networked storage. The `scaleIO` volume plugin allows deployed pods to access existing ScaleIO volumes. For information about dynamically provisioning new volumes for persistent volume claims, see [ScaleIO persistent volumes](#).

Note: You must have an existing ScaleIO cluster already setup and running with the volumes created before you can use them.

The following example is a Pod configuration with ScaleIO:

```
apiVersion: v1
kind: Pod
metadata:
  name: pod-0
spec:
  containers:
    - image: k8s.gcr.io/test-webserver
      name: pod-0
      volumeMounts:
        - mountPath: /test-pd
          name: vol-0
  volumes:
    - name: vol-0
  scaleIO:
    gateway: https://localhost:443/api
    system: scaleio
    protectionDomain: sd0
    storagePool: spl
    volumeName: vol-0
    secretRef:
```

```
  name: sio-secret
  fsType: xfs
```

For further details, see the [ScaleIO](#) examples.

secret

A *secret* volume is used to pass sensitive information, such as passwords, to Pods. You can store secrets in the Kubernetes API and mount them as files for use by pods without coupling to Kubernetes directly. *secret* volumes are backed by tmpfs (a RAM-backed filesystem) so they are never written to non-volatile storage.

Note: You must create a Secret in the Kubernetes API before you can use it.

Note: A container using a Secret as a [subPath](#) volume mount will not receive Secret updates.

For more details, see [Configuring Secrets](#).

storageOS

A *storageos* volume allows an existing [StorageOS](#) volume to mount into your Pod.

StorageOS runs as a container within your Kubernetes environment, making local or attached storage accessible from any node within the Kubernetes cluster. Data can be replicated to protect against node failure. Thin provisioning and compression can improve utilization and reduce cost.

At its core, StorageOS provides block storage to containers, accessible from a file system.

The StorageOS Container requires 64-bit Linux and has no additional dependencies. A free developer license is available.

Caution: You must run the StorageOS container on each node that wants to access StorageOS volumes or that will contribute storage capacity to the pool. For installation instructions, consult the [StorageOS documentation](#).

The following example is a Pod configuration with StorageOS:

```
apiVersion: v1
kind: Pod
metadata:
  labels:
    name: redis
    role: master
  name: test-storageos-redis
spec:
  containers:
```

```

- name: master
  image: kubernetes/redis:v1
  env:
    - name: MASTER
      value: "true"
  ports:
    - containerPort: 6379
  volumeMounts:
    - mountPath: /redis-master-data
      name: redis-data
  volumes:
    - name: redis-data
      storageos:
        # The `redis-vol01` volume must already exist within
        StorageOS in the `default` namespace.
        volumeName: redis-vol01
        fsType: ext4

```

For more information about StorageOS, dynamic provisioning, and PersistentVolumeClaims, see the [StorageOS examples](#).

vsphereVolume

Note: You must configure the Kubernetes vSphere Cloud Provider. For cloudprovider configuration, refer to the [vSphere Getting Started guide](#).

A `vsphereVolume` is used to mount a vSphere VMDK volume into your Pod. The contents of a volume are preserved when it is unmounted. It supports both VMFS and VSAN datastore.

Note: You must create vSphere VMDK volume using one of the following methods before using with a Pod.

Creating a VMDK volume

Choose one of the following methods to create a VMDK.

- [Create using vmkfstools](#)
- [Create using vmware-vdiskmanager](#)

First ssh into ESX, then use the following command to create a VMDK:

```
vmkfstools -c 2G /vmfs/volumes/DatastoreName/volumes/myDisk.vmdk
```

Use the following command to create a VMDK:

```
vmware-vdiskmanager -c -t 0 -s 40GB -a lsilogic myDisk.vmdk
```

vSphere VMDK configuration example

```
apiVersion: v1
kind: Pod
metadata:
  name: test-vmdk
spec:
  containers:
    - image: k8s.gcr.io/test-webserver
      name: test-container
      volumeMounts:
        - mountPath: /test-vmdk
          name: test-volume
  volumes:
    - name: test-volume
      # This VMDK volume must already exist.
      vsphereVolume:
        volumePath: "[DatastoreName] volumes/myDisk"
        fsType: ext4
```

For more information, see the [vSphere volume](#) examples.

vSphere CSI migration

FEATURE STATE: Kubernetes v1.19 [beta]

The CSIMigration feature for `vsphereVolume`, when enabled, redirects all plugin operations from the existing in-tree plugin to the `csi.vsphere.vmware.com CSI` driver. In order to use this feature, the [vSphere CSI driver](#) must be installed on the cluster and the `CSIMigration` and `CSIMigrationSphere feature gates` must be enabled.

This also requires minimum vSphere vCenter/ESXi Version to be 7.0u1 and minimum HW Version to be VM version 15.

Note:

The following `StorageClass` parameters from the built-in `vsphereVolume` plugin are not supported by the vSphere CSI driver:

- `diskformat`
- `hostfailuresetotolerate`
- `forceprovisioning`
- `cachereservation`
- `diskstripes`
- `objectspacereservation`
- `iopslimit`

Existing volumes created using these parameters will be migrated to the vSphere CSI driver, but new volumes created by the vSphere CSI driver will not be honoring these parameters.

vSphere CSI migration complete

FEATURE STATE: Kubernetes v1.19 [beta]

To turn off the `vsphereVolume` plugin from being loaded by the controller manager and the kubelet, you need to set this feature flag to true. You must install a `csi.vsphere.vmware.com` [CSI](#) driver on all worker nodes.

Using subPath

Sometimes, it is useful to share one volume for multiple uses in a single pod. The `volumeMounts.subPath` property specifies a sub-path inside the referenced volume instead of its root.

The following example shows how to configure a Pod with a LAMP stack (Linux Apache MySQL PHP) using a single, shared volume. This sample `subPath` configuration is not recommended for production use.

The PHP application's code and assets map to the volume's `html` folder and the MySQL database is stored in the volume's `mysql` folder. For example:

```
apiVersion: v1
kind: Pod
metadata:
  name: my-lamp-site
spec:
  containers:
    - name: mysql
      image: mysql
      env:
        - name: MYSQL_ROOT_PASSWORD
          value: "rootpasswd"
      volumeMounts:
        - mountPath: /var/lib/mysql
          name: site-data
          subPath: mysql
    - name: php
      image: php:7.0-apache
      volumeMounts:
        - mountPath: /var/www/html
          name: site-data
          subPath: html
  volumes:
    - name: site-data
      persistentVolumeClaim:
        claimName: my-lamp-site-data
```

Using subPath with expanded environment variables

FEATURE STATE: Kubernetes v1.17 [stable]

Use the `subPathExpr` field to construct `subPath` directory names from downward API environment variables. The `subPath` and `subPathExpr` properties are mutually exclusive.

In this example, a Pod uses `subPathExpr` to create a directory `pod1` within the `hostPath` volume `/var/log/pods`. The `hostPath` volume takes the Pod name from the downwardAPI. The host directory `/var/log/pods/pod1` is mounted at `/logs` in the container.

```
apiVersion: v1
kind: Pod
metadata:
  name: pod1
spec:
  containers:
    - name: container1
      env:
        - name: POD_NAME
          valueFrom:
            fieldRef:
              apiVersion: v1
              fieldPath: metadata.name
      image: busybox
      command: [ "sh", "-c", "while [ true ]; do echo 'Hello'; sleep 10; done | tee -a /logs/hello.txt" ]
      volumeMounts:
        - name: workdir1
          mountPath: /logs
          subPathExpr: ${POD_NAME}
    restartPolicy: Never
    volumes:
      - name: workdir1
        hostPath:
          path: /var/log/pods
```

Resources

The storage media (such as Disk or SSD) of an `emptyDir` volume is determined by the medium of the filesystem holding the kubelet root dir (typically `/var/lib/kubelet`). There is no limit on how much space an `emptyDir` or `hostPath` volume can consume, and no isolation between containers or between pods.

To learn about requesting space using a resource specification, see [how to manage resources](#).

Out-of-tree volume plugins

The out-of-tree volume plugins include [Container Storage Interface](#) (CSI) and FlexVolume. These plugins enable storage vendors to create custom

storage plugins without adding their plugin source code to the Kubernetes repository.

Previously, all volume plugins were "in-tree". The "in-tree" plugins were built, linked, compiled, and shipped with the core Kubernetes binaries. This meant that adding a new storage system to Kubernetes (a volume plugin) required checking code into the core Kubernetes code repository.

Both CSI and FlexVolume allow volume plugins to be developed independent of the Kubernetes code base, and deployed (installed) on Kubernetes clusters as extensions.

For storage vendors looking to create an out-of-tree volume plugin, please refer to the [volume plugin FAQ](#).

csi

[Container Storage Interface](#) (CSI) defines a standard interface for container orchestration systems (like Kubernetes) to expose arbitrary storage systems to their container workloads.

Please read the [CSI design proposal](#) for more information.

Note: Support for CSI spec versions 0.2 and 0.3 are deprecated in Kubernetes v1.13 and will be removed in a future release.

Note: CSI drivers may not be compatible across all Kubernetes releases. Please check the specific CSI driver's documentation for supported deployments steps for each Kubernetes release and a compatibility matrix.

Once a CSI compatible volume driver is deployed on a Kubernetes cluster, users may use the `csi` volume type to attach or mount the volumes exposed by the CSI driver.

A csi volume can be used in a Pod in three different ways:

- through a reference to a [PersistentVolumeClaim](#)
- with a [generic ephemeral volume](#) (alpha feature)
- with a [CSI ephemeral volume](#) if the driver supports that (beta feature)

The following fields are available to storage administrators to configure a CSI persistent volume:

- **driver:** A string value that specifies the name of the volume driver to use. This value must correspond to the value returned in the `GetPluginInfoResponse` by the CSI driver as defined in the [CSI spec](#). It is used by Kubernetes to identify which CSI driver to call out to, and by CSI driver components to identify which PV objects belong to the CSI driver.
- **volumeHandle:** A string value that uniquely identifies the volume. This value must correspond to the value returned in the `volume.id` field of the `CreateVolumeResponse` by the CSI driver as defined in the [CSI](#)

[*spec*](#). The value is passed as `volume_id` on all calls to the CSI volume driver when referencing the volume.

- `readOnly`: An optional boolean value indicating whether the volume is to be "ControllerPublished" (attached) as read only. Default is false. This value is passed to the CSI driver via the `readonly` field in the `ControllerPublishVolumeRequest`.
- `fsType`: If the PV's `VolumeMode` is `Filesystem` then this field may be used to specify the filesystem that should be used to mount the volume. If the volume has not been formatted and formatting is supported, this value will be used to format the volume. This value is passed to the CSI driver via the `VolumeCapability` field of `ControllerPublishVolumeRequest`, `NodeStageVolumeRequest`, and `NodePublishVolumeRequest`.
- `volumeAttributes`: A map of string to string that specifies static properties of a volume. This map must correspond to the map returned in the `volume.attributes` field of the `CreateVolumeResponse` by the CSI driver as defined in the [*CSI spec*](#). The map is passed to the CSI driver via the `volume_context` field in the `ControllerPublishVolumeRequest`, `NodeStageVolumeRequest`, and `NodePublishVolumeRequest`.
- `controllerPublishSecretRef`: A reference to the secret object containing sensitive information to pass to the CSI driver to complete the CSI `ControllerPublishVolume` and `ControllerUnpublishVolume` calls. This field is optional, and may be empty if no secret is required. If the Secret contains more than one secret, all secrets are passed.
- `nodeStageSecretRef`: A reference to the secret object containing sensitive information to pass to the CSI driver to complete the CSI `NodeStageVolume` call. This field is optional, and may be empty if no secret is required. If the Secret contains more than one secret, all secrets are passed.
- `nodePublishSecretRef`: A reference to the secret object containing sensitive information to pass to the CSI driver to complete the CSI `NodePublishVolume` call. This field is optional, and may be empty if no secret is required. If the secret object contains more than one secret, all secrets are passed.

CSI raw block volume support

FEATURE STATE: Kubernetes v1.18 [stable]

Vendors with external CSI drivers can implement raw block volume support in Kubernetes workloads.

You can set up your [`PersistentVolume/PersistentVolumeClaim` with raw block volume support](#) as usual, without any CSI specific changes.

CSI ephemeral volumes

FEATURE STATE: Kubernetes v1.16 [beta]

You can directly configure CSI volumes within the Pod specification. Volumes specified in this way are ephemeral and do not persist across pod restarts. See [`Ephemeral Volumes`](#) for more information.

For more information on how to develop a CSI driver, refer to the [kubernetes-csi documentation](#)

Migrating to CSI drivers from in-tree plugins

FEATURE STATE: Kubernetes v1.17 [beta]

The *CSIMigration* feature, when enabled, directs operations against existing in-tree plugins to corresponding CSI plugins (which are expected to be installed and configured). As a result, operators do not have to make any configuration changes to existing Storage Classes, PersistentVolumes or PersistentVolumeClaims (referring to in-tree plugins) when transitioning to a CSI driver that supersedes an in-tree plugin.

The operations and features that are supported include: provisioning/delete, attach/detach, mount/unmount and resizing of volumes.

In-tree plugins that support *CSIMigration* and have a corresponding CSI driver implemented are listed in [Types of Volumes](#).

flexVolume

FlexVolume is an out-of-tree plugin interface that has existed in Kubernetes since version 1.2 (before CSI). It uses an exec-based model to interface with drivers. The *FlexVolume* driver binaries must be installed in a pre-defined volume plugin path on each node and in some cases the control plane nodes as well.

Pods interact with FlexVolume drivers through the `flexvolume` in-tree volume plugin. For more details, see the [FlexVolume](#) examples.

Mount propagation

Mount propagation allows for sharing volumes mounted by a container to other containers in the same pod, or even to other pods on the same node.

Mount propagation of a volume is controlled by the `mountPropagation` field in `Container.volumeMounts`. Its values are:

- ***None** - This volume mount will not receive any subsequent mounts that are mounted to this volume or any of its subdirectories by the host. In similar fashion, no mounts created by the container will be visible on the host. This is the default mode.*

This mode is equal to private mount propagation as described in the [Linux kernel documentation](#)

- ***HostToContainer** - This volume mount will receive all subsequent mounts that are mounted to this volume or any of its subdirectories.*

In other words, if the host mounts anything inside the volume mount, the container will see it mounted there.

*Similarly, if any Pod with **Bidirectional** mount propagation to the same volume mounts anything there, the container with **HostToContainer** mount propagation will see it.*

This mode is equal to `rslave` mount propagation as described in the [Linux kernel documentation](#)

- **Bidirectional** - This volume mount behaves the same as the **HostToContainer** mount. In addition, all volume mounts created by the container will be propagated back to the host and to all containers of all pods that use the same volume.

A typical use case for this mode is a Pod with a `FlexVolume` or `CSI` driver or a Pod that needs to mount something on the host using a `hostPath` volume.

This mode is equal to `rshared` mount propagation as described in the [Linux kernel documentation](#)

Warning: *Bidirectional* mount propagation can be dangerous. It can damage the host operating system and therefore it is allowed only in privileged containers.

Familiarity with Linux kernel behavior is strongly recommended. In addition, any volume mounts created by containers in pods must be destroyed (unmounted) by the containers on termination.

Configuration

Before mount propagation can work properly on some deployments (CoreOS, RedHat/Centos, Ubuntu) mount share must be configured correctly in Docker as shown below.

Edit your Docker's `systemd` service file. Set `MountFlags` as follows:

MountFlags=shared

Or, remove `MountFlags=slave` if present. Then restart the Docker daemon:

```
sudo systemctl daemon-reload  
sudo systemctl restart docker
```

What's next

Follow an example of [deploying WordPress and MySQL with Persistent Volumes](#).

Feedback

Was this page helpful?

Yes No

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

Last modified November 05, 2020 at 11:31 AM PST: [Document](#)

[SizeMemoryBackedVolumes feature for 1.20 \(af239416b\)](#)

[Edit this page](#) [Create child page](#) [Create an issue](#)

- [Background](#)
- [Types of Volumes](#)
 - [awsElasticBlockStore](#)
 - [azureDisk](#)
 - [azureFile](#)
 - [cephfs](#)
 - [cinder](#)
 - [configMap](#)
 - [downwardAPI](#)
 - [emptyDir](#)
 - [fc \(fibre channel\)](#)
 - [flocker \(deprecated\)](#)
 - [gcePersistentDisk](#)
 - [gitRepo \(deprecated\)](#)
 - [glusterfs](#)
 - [hostPath](#)
 - [iscsi](#)
 - [local](#)
 - [nfs](#)
 - [persistentVolumeClaim](#)
 - [portworxVolume](#)
 - [projected](#)
 - [quobyte](#)
 - [rbd](#)
 - [scaleIO \(deprecated\)](#)
 - [secret](#)
 - [storageOS](#)
 - [vsphereVolume](#)
- [Using subPath](#)
 - [Using subPath with expanded environment variables](#)
- [Resources](#)
- [Out-of-tree volume plugins](#)
 - [csi](#)
 - [flexVolume](#)
- [Mount propagation](#)
 - [Configuration](#)
- [What's next](#)

Persistent Volumes

This document describes the current state of persistent volumes in Kubernetes. Familiarity with [volumes](#) is suggested.

Introduction

Managing storage is a distinct problem from managing compute instances. The PersistentVolume subsystem provides an API for users and administrators that abstracts details of how storage is provided from how it is consumed. To do this, we introduce two new API resources: PersistentVolume and PersistentVolumeClaim.

A PersistentVolume (PV) is a piece of storage in the cluster that has been provisioned by an administrator or dynamically provisioned using [Storage Classes](#). It is a resource in the cluster just like a node is a cluster resource. PVs are volume plugins like Volumes, but have a lifecycle independent of any individual Pod that uses the PV. This API object captures the details of the implementation of the storage, be that NFS, iSCSI, or a cloud-provider-specific storage system.

A PersistentVolumeClaim (PVC) is a request for storage by a user. It is similar to a Pod. Pods consume node resources and PVCs consume PV resources. Pods can request specific levels of resources (CPU and Memory). Claims can request specific size and access modes (e.g., they can be mounted ReadWriteOnce, ReadOnlyMany or ReadWriteMany, see [AccessModes](#)).

While PersistentVolumeClaims allow a user to consume abstract storage resources, it is common that users need PersistentVolumes with varying properties, such as performance, for different problems. Cluster administrators need to be able to offer a variety of PersistentVolumes that differ in more ways than just size and access modes, without exposing users to the details of how those volumes are implemented. For these needs, there is the StorageClass resource.

See the [detailed walkthrough with working examples](#).

Lifecycle of a volume and claim

PVs are resources in the cluster. PVCs are requests for those resources and also act as claim checks to the resource. The interaction between PVs and PVCs follows this lifecycle:

Provisioning

There are two ways PVs may be provisioned: statically or dynamically.

Static

A cluster administrator creates a number of PVs. They carry the details of the real storage, which is available for use by cluster users. They exist in the Kubernetes API and are available for consumption.

Dynamic

When none of the static PVs the administrator created match a user's PersistentVolumeClaim, the cluster may try to dynamically provision a volume specially for the PVC. This provisioning is based on StorageClasses: the PVC must request a [storage class](#) and the administrator must have created and configured that class for dynamic provisioning to occur. Claims that request the class "" effectively disable dynamic provisioning for themselves.

To enable dynamic storage provisioning based on storage class, the cluster administrator needs to enable the DefaultStorageClass [admission controller](#) on the API server. This can be done, for example, by ensuring that DefaultStorageClass is among the comma-delimited, ordered list of values for the --enable-admission-plugins flag of the API server component. For more information on API server command-line flags, check [kube-apiserver](#) documentation.

Binding

A user creates, or in the case of dynamic provisioning, has already created, a PersistentVolumeClaim with a specific amount of storage requested and with certain access modes. A control loop in the master watches for new PVCs, finds a matching PV (if possible), and binds them together. If a PV was dynamically provisioned for a new PVC, the loop will always bind that PV to the PVC. Otherwise, the user will always get at least what they asked for, but the volume may be in excess of what was requested. Once bound, PersistentVolumeClaim binds are exclusive, regardless of how they were bound. A PVC to PV binding is a one-to-one mapping, using a ClaimRef which is a bi-directional binding between the PersistentVolume and the PersistentVolumeClaim.

Claims will remain unbound indefinitely if a matching volume does not exist. Claims will be bound as matching volumes become available. For example, a cluster provisioned with many 50Gi PVs would not match a PVC requesting 100Gi. The PVC can be bound when a 100Gi PV is added to the cluster.

Using

Pods use claims as volumes. The cluster inspects the claim to find the bound volume and mounts that volume for a Pod. For volumes that support multiple access modes, the user specifies which mode is desired when using their claim as a volume in a Pod.

Once a user has a claim and that claim is bound, the bound PV belongs to the user for as long as they need it. Users schedule Pods and access their claimed PVs by including a persistentVolumeClaim section in a Pod's volumes block. See [Claims As Volumes](#) for more details on this.

Storage Object in Use Protection

The purpose of the Storage Object in Use Protection feature is to ensure that PersistentVolumeClaims (PVCs) in active use by a Pod and PersistentVolume (PVs) that are bound to PVCs are not removed from the system, as this may result in data loss.

Note: PVC is in active use by a Pod when a Pod object exists that is using the PVC.

If a user deletes a PVC in active use by a Pod, the PVC is not removed immediately. PVC removal is postponed until the PVC is no longer actively used by any Pods. Also, if an admin deletes a PV that is bound to a PVC, the PV is not removed immediately. PV removal is postponed until the PV is no longer bound to a PVC.

You can see that a PVC is protected when the PVC's status is Terminating and the Finalizers list includes kubernetes.io/pvc-protection:

```
kubectl describe pvc hostpath
Name:          hostpath
Namespace:     default
StorageClass:  example-hostpath
Status:        Terminating
Volume:
Labels:         <none>
Annotations:   volume.beta.kubernetes.io/storage-class=example-
hostpath           volume.beta.kubernetes.io/storage-provisioner=exam-
ple.com/hostpath
Finalizers:    [kubernetes.io/pvc-protection]
...
...
```

You can see that a PV is protected when the PV's status is Terminating and the Finalizers list includes kubernetes.io/pv-protection too:

```
kubectl describe pv task-pv-volume
Name:          task-pv-volume
Labels:        type=local
Annotations:   <none>
Finalizers:   [kubernetes.io/pv-protection]
StorageClass: standard
Status:        Terminating
Claim:
Reclaim Policy: Delete
Access Modes:  RWO
Capacity:      1Gi
Message:
Source:
  Type:       HostPath (bare host directory volume)
  Path:      /tmp/data
...
```

<code>HostPathType:</code>	
<code>Events:</code>	<code><none></code>

Reclaiming

When a user is done with their volume, they can delete the PVC objects from the API that allows reclamation of the resource. The reclaim policy for a PersistentVolume tells the cluster what to do with the volume after it has been released of its claim. Currently, volumes can either be Retained, Recycled, or Deleted.

Retain

The Retain reclaim policy allows for manual reclamation of the resource. When the PersistentVolumeClaim is deleted, the PersistentVolume still exists and the volume is considered "released". But it is not yet available for another claim because the previous claimant's data remains on the volume. An administrator can manually reclaim the volume with the following steps.

1. Delete the PersistentVolume. The associated storage asset in external infrastructure (such as an AWS EBS, GCE PD, Azure Disk, or Cinder volume) still exists after the PV is deleted.
2. Manually clean up the data on the associated storage asset accordingly.
3. Manually delete the associated storage asset, or if you want to reuse the same storage asset, create a new PersistentVolume with the storage asset definition.

Delete

For volume plugins that support the Delete reclaim policy, deletion removes both the PersistentVolume object from Kubernetes, as well as the associated storage asset in the external infrastructure, such as an AWS EBS, GCE PD, Azure Disk, or Cinder volume. Volumes that were dynamically provisioned inherit the [reclaim policy of their StorageClass](#), which defaults to Delete. The administrator should configure the StorageClass according to users' expectations; otherwise, the PV must be edited or patched after it is created. See [Change the Reclaim Policy of a PersistentVolume](#).

Recycle

Warning: The Recycle reclaim policy is deprecated. Instead, the recommended approach is to use dynamic provisioning.

If supported by the underlying volume plugin, the Recycle reclaim policy performs a basic scrub (`rm -rf /thevolume/`) on the volume and makes it available again for a new claim.*

However, an administrator can configure a custom recycler Pod template using the Kubernetes controller manager command line arguments as described in the [reference](#). The custom recycler Pod template must contain a volumes specification, as shown in the example below:

```

apiVersion: v1
kind: Pod
metadata:
  name: pv-recycler
  namespace: default
spec:
  restartPolicy: Never
  volumes:
  - name: vol
    hostPath:
      path: /any/path/it/will/be/replaced
  containers:
  - name: pv-recycler
    image: "k8s.gcr.io/busybox"
    command: ["/bin/sh", "-c", "test -e /scrub && rm -rf /scrub/..?* /scrub/.[!.]* /scrub/* && test -z \"$(ls -A /scrub)\" || exit 1"]
    volumeMounts:
    - name: vol
      mountPath: /scrub

```

However, the particular path specified in the custom recycler Pod template in the `volumes` part is replaced with the particular path of the volume that is being recycled.

Reserving a PersistentVolume

The control plane can [bind PersistentVolumeClaims to matching PersistentVolumes](#) in the cluster. However, if you want a PVC to bind to a specific PV, you need to pre-bind them.

By specifying a PersistentVolume in a PersistentVolumeClaim, you declare a binding between that specific PV and PVC. If the PersistentVolume exists and has not reserved PersistentVolumeClaims through its `claimRef` field, then the PersistentVolume and PersistentVolumeClaim will be bound.

The binding happens regardless of some volume matching criteria, including node affinity. The control plane still checks that [storage class](#), access modes, and requested storage size are valid.

```

apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: foo-pvc
  namespace: foo
spec:
  storageClassName: "" # Empty string must be explicitly set
  otherwise default StorageClass will be set
  volumeName: foo-pv

```

This method does not guarantee any binding privileges to the PersistentVolume. If other PersistentVolumeClaims could use the PV that you specify, you first need to reserve that storage volume. Specify the relevant PersistentVolumeClaim in the claimRef field of the PV so that other PVCs can not bind to it.

```
apiVersion: v1
kind: PersistentVolume
metadata:
  name: foo-pv
spec:
  storageClassName: ""
  claimRef:
    name: foo-pvc
    namespace: foo
  ...
```

This is useful if you want to consume PersistentVolumes that have their claimPolicy set to Retain, including cases where you are reusing an existing PV.

Expanding Persistent Volumes Claims

FEATURE STATE: Kubernetes v1.11 [beta]

Support for expanding PersistentVolumeClaims (PVCs) is now enabled by default. You can expand the following types of volumes:

- gcePersistentDisk
- awsElasticBlockStore
- Cinder
- glusterfs
- rbd
- Azure File
- Azure Disk
- Portworx
- FlexVolumes
- CSI

You can only expand a PVC if its storage class's allowVolumeExpansion field is set to true.

```
apiVersion: storage.k8s.io/v1
kind: StorageClass
metadata:
  name: gluster-vol-default
provisioner: kubernetes.io/glusterfs
parameters:
  resturl: "http://192.168.10.100:8080"
  restuser: ""
  secretNamespace: ""
  secretName: ""
allowVolumeExpansion: true
```

To request a larger volume for a PVC, edit the PVC object and specify a larger size. This triggers expansion of the volume that backs the underlying PersistentVolume. A new PersistentVolume is never created to satisfy the claim. Instead, an existing volume is resized.

CSI Volume expansion

FEATURE STATE: Kubernetes v1.16 [beta]

Support for expanding CSI volumes is enabled by default but it also requires a specific CSI driver to support volume expansion. Refer to documentation of the specific CSI driver for more information.

Resizing a volume containing a file system

You can only resize volumes containing a file system if the file system is XFS, Ext3, or Ext4.

When a volume contains a file system, the file system is only resized when a new Pod is using the PersistentVolumeClaim in ReadWrite mode. File system expansion is either done when a Pod is starting up or when a Pod is running and the underlying file system supports online expansion.

FlexVolumes allow resize if the driver is set with the RequiresFSResize capability to true. The FlexVolume can be resized on Pod restart.

Resizing an in-use PersistentVolumeClaim

FEATURE STATE: Kubernetes v1.15 [beta]

Note: Expanding in-use PVCs is available as beta since Kubernetes 1.15, and as alpha since 1.11. The `ExpandInUsePersistentVolume`s feature must be enabled, which is the case automatically for many clusters for beta features. Refer to the [feature gate](#) documentation for more information.

In this case, you don't need to delete and recreate a Pod or deployment that is using an existing PVC. Any in-use PVC automatically becomes available to its Pod as soon as its file system has been expanded. This feature has no effect on PVCs that are not in use by a Pod or deployment. You must create a Pod that uses the PVC before the expansion can complete.

Similar to other volume types - FlexVolume volumes can also be expanded when in-use by a Pod.

Note: FlexVolume resize is possible only when the underlying driver supports resize.

Note: Expanding EBS volumes is a time-consuming operation. Also, there is a per-volume quota of one modification every 6 hours.

Recovering from Failure when Expanding Volumes

If expanding underlying storage fails, the cluster administrator can manually recover the Persistent Volume Claim (PVC) state and cancel the resize requests. Otherwise, the resize requests are continuously retried by the controller without administrator intervention.

1. *Mark the PersistentVolume(PV) that is bound to the PersistentVolumeClaim(PVC) with Retain reclaim policy.*
2. *Delete the PVC. Since PV has Retain reclaim policy - we will not lose any data when we recreate the PVC.*
3. *Delete the claimRef entry from PV specs, so as new PVC can bind to it. This should make the PV Available.*
4. *Re-create the PVC with smaller size than PV and set volumeName field of the PVC to the name of the PV. This should bind new PVC to existing PV.*
5. *Don't forget to restore the reclaim policy of the PV.*

Types of Persistent Volumes

PersistentVolume types are implemented as plugins. Kubernetes currently supports the following plugins:

- [awsElasticBlockStore](#) - AWS Elastic Block Store (EBS)
- [azureDisk](#) - Azure Disk
- [azureFile](#) - Azure File
- [cephfs](#) - CephFS volume
- [cinder](#) - Cinder (OpenStack block storage) (**deprecated**)
- [csi](#) - Container Storage Interface (CSI)
- [fc](#) - Fibre Channel (FC) storage
- [flexVolume](#) - FlexVolume
- [flocker](#) - Flocker storage
- [gcePersistentDisk](#) - GCE Persistent Disk
- [glusterfs](#) - Glusterfs volume
- [hostPath](#) - HostPath volume (for single node testing only; WILL NOT WORK in a multi-node cluster; consider using `local` volume instead)
- [iscsi](#) - iSCSI (SCSI over IP) storage
- [local](#) - local storage devices mounted on nodes.
- [nfs](#) - Network File System (NFS) storage
- [photonPersistentDisk](#) - Photon controller persistent disk. (This volume type no longer works since the removal of the corresponding cloud provider.)
- [portworxVolume](#) - Portworx volume
- [quobyte](#) - Quobyte volume
- [rbd](#) - Rados Block Device (RBD) volume
- [scaleIO](#) - ScaleIO volume (**deprecated**)
- [storageos](#) - StorageOS volume
- [vsphereVolume](#) - vSphere VMDK volume

Persistent Volumes

Each PV contains a spec and status, which is the specification and status of the volume. The name of a PersistentVolume object must be a valid [DNS subdomain name](#).

```
apiVersion: v1
kind: PersistentVolume
metadata:
  name: pv0003
spec:
  capacity:
    storage: 5Gi
  volumeMode: Filesystem
  accessModes:
    - ReadWriteOnce
  persistentVolumeReclaimPolicy: Recycle
  storageClassName: slow
  mountOptions:
    - hard
    - nfsvers=4.1
  nfs:
    path: /tmp
    server: 172.17.0.2
```

Note: Helper programs relating to the volume type may be required for consumption of a PersistentVolume within a cluster. In this example, the PersistentVolume is of type NFS and the helper program `/sbin/mount.nfs` is required to support the mounting of NFS filesystems.

Capacity

Generally, a PV will have a specific storage capacity. This is set using the PV's capacity attribute. See the Kubernetes [Resource Model](#) to understand the units expected by capacity.

Currently, storage size is the only resource that can be set or requested. Future attributes may include IOPS, throughput, etc.

Volume Mode

FEATURE STATE: Kubernetes v1.18 [stable]

Kubernetes supports two volumeModes of PersistentVolumes: `Filesystem` and `Block`.

`volumeMode` is an optional API parameter. `Filesystem` is the default mode used when `volumeMode` parameter is omitted.

A volume with `volumeMode: Filesystem` is mounted into Pods into a directory. If the volume is backed by a block device and the device is empty,

Kubernetes creates a filesystem on the device before mounting it for the first time.

You can set the value of `volumeMode` to `Block` to use a volume as a raw block device. Such volume is presented into a Pod as a block device, without any filesystem on it. This mode is useful to provide a Pod the fastest possible way to access a volume, without any filesystem layer between the Pod and the volume. On the other hand, the application running in the Pod must know how to handle a raw block device. See [Raw Block Volume Support](#) for an example on how to use a volume with `volumeMode: Block` in a Pod.

Access Modes

A `PersistentVolume` can be mounted on a host in any way supported by the resource provider. As shown in the table below, providers will have different capabilities and each PV's access modes are set to the specific modes supported by that particular volume. For example, NFS can support multiple read/write clients, but a specific NFS PV might be exported on the server as read-only. Each PV gets its own set of access modes describing that specific PV's capabilities.

The access modes are:

- `ReadWriteOnce` -- the volume can be mounted as read-write by a single node
- `ReadOnlyMany` -- the volume can be mounted read-only by many nodes
- `ReadWriteMany` -- the volume can be mounted as read-write by many nodes

In the CLI, the access modes are abbreviated to:

- `RWO` - `ReadWriteOnce`
- `ROX` - `ReadOnlyMany`
- `RWX` - `ReadWriteMany`

Important! A volume can only be mounted using one access mode at a time, even if it supports many. For example, a `GCEPersistentDisk` can be mounted as `ReadWriteOnce` by a single node or `ReadOnlyMany` by many nodes, but not at the same time.

Volume Plugin	<code>ReadWriteOnce</code>	<code>ReadOnlyMany</code>	<code>ReadWriteMany</code>
AWSElasticBlockStore	“	-	-
AzureFile	“	“	“
AzureDisk	“	-	-
CephFS	“	“	“
Cinder	“	-	-
CSI	depends on the driver	depends on the driver	depends on the driver
FC	“	“	-
FlexVolume	“	“	depends on the driver

Volume Plugin	ReadWriteOnce	ReadOnlyMany	ReadWriteMany
Flocker	“	-	-
GCEPersistentDisk	“	“	-
Glusterfs	“	“	“
HostPath	“	-	-
iSCSI	“	“	-
Quobyte	“	“	“
NFS	“	“	“
RBD	“	“	-
VsphereVolume	“	-	- (works when Pods are collocated)
PortworxVolume	“	-	“
ScaleIO	“	“	-
StorageOS	“	-	-

Class

A PV can have a class, which is specified by setting the `storageClassName` attribute to the name of a [StorageClass](#). A PV of a particular class can only be bound to PVCs requesting that class. A PV with no `storageClassName` has no class and can only be bound to PVCs that request no particular class.

In the past, the annotation `volume.beta.kubernetes.io/storage-class` was used instead of the `storageClassName` attribute. This annotation is still working; however, it will become fully deprecated in a future Kubernetes release.

Reclaim Policy

Current reclaim policies are:

- Retain -- manual reclamation
- Recycle -- basic scrub (`rm -rf /thevolume/*`)
- Delete -- associated storage asset such as AWS EBS, GCE PD, Azure Disk, or OpenStack Cinder volume is deleted

Currently, only NFS and HostPath support recycling. AWS EBS, GCE PD, Azure Disk, and Cinder volumes support deletion.

Mount Options

A Kubernetes administrator can specify additional mount options for when a Persistent Volume is mounted on a node.

Note: Not all Persistent Volume types support mount options.

The following volume types support mount options:

- AWSElasticBlockStore

- *AzureDisk*
- *AzureFile*
- *CephFS*
- *Cinder (OpenStack block storage)*
- *GCEPersistentDisk*
- *Glusterfs*
- *NFS*
- *Quobyte Volumes*
- *RBD (Ceph Block Device)*
- *StorageOS*
- *VsphereVolume*
- *iSCSI*

Mount options are not validated, so mount will simply fail if one is invalid.

In the past, the annotation `volume.beta.kubernetes.io/mount-options` was used instead of the `mountOptions` attribute. This annotation is still working; however, it will become fully deprecated in a future Kubernetes release.

Node Affinity

Note: For most volume types, you do not need to set this field. It is automatically populated for [AWS EBS](#), [GCE PD](#) and [Azure Disk](#) volume block types. You need to explicitly set this for [local](#) volumes.

A PV can specify [node affinity](#) to define constraints that limit what nodes this volume can be accessed from. Pods that use a PV will only be scheduled to nodes that are selected by the node affinity.

Phase

A volume will be in one of the following phases:

- Available -- a free resource that is not yet bound to a claim
- Bound -- the volume is bound to a claim
- Released -- the claim has been deleted, but the resource is not yet reclaimed by the cluster
- Failed -- the volume has failed its automatic reclamation

The CLI will show the name of the PVC bound to the PV.

PersistentVolumeClaims

Each PVC contains a spec and status, which is the specification and status of the claim. The name of a PersistentVolumeClaim object must be a valid [DNS subdomain name](#).

```
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
```

```

name: myclaim
spec:
  accessModes:
    - ReadWriteOnce
  volumeMode: Filesystem
  resources:
    requests:
      storage: 8Gi
  storageClassName: slow
  selector:
    matchLabels:
      release: "stable"
    matchExpressions:
      - {key: environment, operator: In, values: [dev]}

```

Access Modes

Claims use the same conventions as volumes when requesting storage with specific access modes.

Volume Modes

Claims use the same convention as volumes to indicate the consumption of the volume as either a filesystem or block device.

Resources

Claims, like Pods, can request specific quantities of a resource. In this case, the request is for storage. The same [resource model](#) applies to both volumes and claims.

Selector

Claims can specify a [label selector](#) to further filter the set of volumes. Only the volumes whose labels match the selector can be bound to the claim. The selector can consist of two fields:

- *matchLabels* - the volume must have a label with this value
- *matchExpressions* - a list of requirements made by specifying key, list of values, and operator that relates the key and values. Valid operators include *In*, *NotIn*, *Exists*, and *DoesNotExist*.

*All of the requirements, from both *matchLabels* and *matchExpressions*, are ANDed together - they must all be satisfied in order to match.*

Class

*A claim can request a particular class by specifying the name of a [StorageClass](#) using the attribute *storageClassName*. Only PVs of the requested class, ones with the same *storageClassName* as the PVC, can be bound to the PVC.*

PVCs don't necessarily have to request a class. A PVC with its `storageClassName` set equal to `" "` is always interpreted to be requesting a PV with no class, so it can only be bound to PVs with no class (no annotation or one set equal to `" "`). A PVC with no `storageClassName` is not quite the same and is treated differently by the cluster, depending on whether the [DefaultStorageClass admission plugin](#) is turned on.

- If the admission plugin is turned on, the administrator may specify a default StorageClass. All PVCs that have no `storageClassName` can be bound only to PVs of that default. Specifying a default StorageClass is done by setting the annotation `storageclass.kubernetes.io/is-default-class` equal to `true` in a StorageClass object. If the administrator does not specify a default, the cluster responds to PVC creation as if the admission plugin were turned off. If more than one default is specified, the admission plugin forbids the creation of all PVCs.
- If the admission plugin is turned off, there is no notion of a default StorageClass. All PVCs that have no `storageClassName` can be bound only to PVs that have no class. In this case, the PVCs that have no `storageClassName` are treated the same way as PVCs that have their `storageClassName` set to `" "`.

Depending on installation method, a default StorageClass may be deployed to a Kubernetes cluster by addon manager during installation.

When a PVC specifies a `selector` in addition to requesting a StorageClass, the requirements are ANDed together: only a PV of the requested class and with the requested labels may be bound to the PVC.

Note: Currently, a PVC with a non-empty `selector` can't have a PV dynamically provisioned for it.

In the past, the annotation `volume.beta.kubernetes.io/storage-class` was used instead of `storageClassName` attribute. This annotation is still working; however, it won't be supported in a future Kubernetes release.

Claims As Volumes

Pods access storage by using the claim as a volume. Claims must exist in the same namespace as the Pod using the claim. The cluster finds the claim in the Pod's namespace and uses it to get the PersistentVolume backing the claim. The volume is then mounted to the host and into the Pod.

```
apiVersion: v1
kind: Pod
metadata:
  name: mypod
spec:
  containers:
    - name: myfrontend
      image: nginx
      volumeMounts:
```

```
- mountPath: "/var/www/html"
  name: mypd
volumes:
- name: mypd
  persistentVolumeClaim:
    claimName: myclaim
```

A Note on Namespaces

PersistentVolumes binds are exclusive, and since PersistentVolumeClaims are namespaced objects, mounting claims with "Many" modes (R0X, RWX) is only possible within one namespace.

Raw Block Volume Support

FEATURE STATE: Kubernetes v1.18 [stable]

The following volume plugins support raw block volumes, including dynamic provisioning where applicable:

- AWSElasticBlockStore
- AzureDisk
- CSI
- FC (Fibre Channel)
- GCEPersistentDisk
- iSCSI
- Local volume
- OpenStack Cinder
- RBD (Ceph Block Device)
- VsphereVolume

PersistentVolume using a Raw Block Volume

```
apiVersion: v1
kind: PersistentVolume
metadata:
  name: block-pv
spec:
  capacity:
    storage: 10Gi
  accessModes:
    - ReadWriteOnce
  volumeMode: Block
  persistentVolumeReclaimPolicy: Retain
  fc:
    targetWWNs: ["50060e801049cf1"]
    lun: 0
    readOnly: false
```

PersistentVolumeClaim requesting a Raw Block Volume

```
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: block-pvc
spec:
  accessModes:
    - ReadWriteOnce
  volumeMode: Block
  resources:
    requests:
      storage: 10Gi
```

Pod specification adding Raw Block Device path in container

```
apiVersion: v1
kind: Pod
metadata:
  name: pod-with-block-volume
spec:
  containers:
    - name: fc-container
      image: fedora:26
      command: ["/bin/sh", "-c"]
      args: [ "tail -f /dev/null" ]
      volumeDevices:
        - name: data
          devicePath: /dev/xvda
  volumes:
    - name: data
      persistentVolumeClaim:
        claimName: block-pvc
```

Note: When adding a raw block device for a Pod, you specify the device path in the container instead of a mount path.

Binding Block Volumes

If a user requests a raw block volume by indicating this using the `volumeMode` field in the `PersistentVolumeClaim` spec, the binding rules differ slightly from previous releases that didn't consider this mode as part of the spec. Listed is a table of possible combinations the user and admin might specify for requesting a raw block device. The table indicates if the volume will be bound or not given the combinations: Volume binding matrix for statically provisioned volumes:

PV volumeMode	PVC volumeMode	Result
unspecified	unspecified	BIND
unspecified	Block	NO BIND

PV volumeMode	PVC volumeMode	Result
unspecified	Filesystem	BIND
Block	unspecified	NO BIND
Block	Block	BIND
Block	Filesystem	NO BIND
Filesystem	Filesystem	BIND
Filesystem	Block	NO BIND
Filesystem	unspecified	BIND

Note: Only statically provisioned volumes are supported for alpha release. Administrators should take care to consider these values when working with raw block devices.

Volume Snapshot and Restore Volume from Snapshot Support

FEATURE STATE: Kubernetes v1.20 [stable]

Volume snapshots only support the out-of-tree CSI volume plugins. For details, see [Volume Snapshots](#). In-tree volume plugins are deprecated. You can read about the deprecated volume plugins in the [Volume Plugin FAQ] (<https://github.com/kubernetes/community/blob/master/sig-storage/volume-plugin-faq.md>).

Create a PersistentVolumeClaim from a Volume Snapshot

```
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: restore-pvc
spec:
  storageClassName: csi-hostpath-sc
  dataSource:
    name: new-snapshot-test
    kind: VolumeSnapshot
    apiGroup: snapshot.storage.k8s.io
  accessModes:
    - ReadWriteOnce
  resources:
    requests:
      storage: 10Gi
```

Volume Cloning

[Volume Cloning](#) only available for CSI volume plugins.

Create PersistentVolumeClaim from an existing PVC

```
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: cloned-pvc
spec:
  storageClassName: my-csi-plugin
  dataSource:
    name: existing-src-pvc-name
    kind: PersistentVolumeClaim
  accessModes:
    - ReadWriteOnce
  resources:
    requests:
      storage: 10Gi
```

Writing Portable Configuration

If you're writing configuration templates or examples that run on a wide range of clusters and need persistent storage, it is recommended that you use the following pattern:

- Include PersistentVolumeClaim objects in your bundle of config (alongside Deployments, ConfigMaps, etc).
- Do not include PersistentVolume objects in the config, since the user instantiating the config may not have permission to create PersistentVolumes.
- Give the user the option of providing a storage class name when instantiating the template.
 - If the user provides a storage class name, put that value into the `persistentVolumeClaim.storageClassName` field. This will cause the PVC to match the right storage class if the cluster has `StorageClasses` enabled by the admin.
 - If the user does not provide a storage class name, leave the `persistentVolumeClaim.storageClassName` field as `nil`. This will cause a PV to be automatically provisioned for the user with the default `StorageClass` in the cluster. Many cluster environments have a default `StorageClass` installed, or administrators can create their own default `StorageClass`.
- In your tooling, watch for PVCs that are not getting bound after some time and surface this to the user, as this may indicate that the cluster has no dynamic storage support (in which case the user should create a matching PV) or the cluster has no storage system (in which case the user cannot deploy config requiring PVCs).

What's next

- Learn more about [Creating a PersistentVolume](#).
- Learn more about [Creating a PersistentVolumeClaim](#).
- Read the [Persistent Storage design document](#).

Reference

- [PersistentVolume](#)
- [PersistentVolumeSpec](#)
- [PersistentVolumeClaim](#)
- [PersistentVolumeClaimSpec](#)

Feedback

Was this page helpful?

Yes No

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

Last modified September 08, 2020 at 4:06 PM PST: [Add links to volumes from persistent volumes \(9ddc0a3ad\)](#)

[Edit this page](#) [Create child page](#) [Create an issue](#)

- [Introduction](#)
- [Lifecycle of a volume and claim](#)
 - [Provisioning](#)
 - [Binding](#)
 - [Using](#)
 - [Storage Object in Use Protection](#)
 - [Reclaiming](#)
 - [Reserving a PersistentVolume](#)
 - [Expanding Persistent Volumes Claims](#)
- [Types of Persistent Volumes](#)
- [Persistent Volumes](#)
 - [Capacity](#)
 - [Volume Mode](#)
 - [Access Modes](#)
 - [Class](#)
 - [Reclaim Policy](#)
 - [Mount Options](#)
 - [Node Affinity](#)
 - [Phase](#)
- [PersistentVolumeClaims](#)
 - [Access Modes](#)
 - [Volume Modes](#)
 - [Resources](#)
 - [Selector](#)

- [Class](#)
- [Claims As Volumes](#)
 - [A Note on Namespaces](#)
- [Raw Block Volume Support](#)
 - [PersistentVolume using a Raw Block Volume](#)
 - [PersistentVolumeClaim requesting a Raw Block Volume](#)
 - [Pod specification adding Raw Block Device path in container](#)
 - [Binding Block Volumes](#)
- [Volume Snapshot and Restore Volume from Snapshot Support](#)
 - [Create a PersistentVolumeClaim from a Volume Snapshot](#)
- [Volume Cloning](#)
 - [Create PersistentVolumeClaim from an existing PVC](#)
- [Writing Portable Configuration](#)
- [What's next](#)
 - [Reference](#)

Volume Snapshots

In Kubernetes, a `VolumeSnapshot` represents a snapshot of a volume on a storage system. This document assumes that you are already familiar with Kubernetes [persistent volumes](#).

Introduction

Similar to how API resources `PersistentVolume` and `PersistentVolumeClaim` are used to provision volumes for users and administrators, `VolumeSnapshotContent` and `VolumeSnapshot` API resources are provided to create volume snapshots for users and administrators.

A `VolumeSnapshotContent` is a snapshot taken from a volume in the cluster that has been provisioned by an administrator. It is a resource in the cluster just like a `PersistentVolume` is a cluster resource.

A `VolumeSnapshot` is a request for snapshot of a volume by a user. It is similar to a `PersistentVolumeClaim`.

`VolumeSnapshotClass` allows you to specify different attributes belonging to a `VolumeSnapshot`. These attributes may differ among snapshots taken from the same volume on the storage system and therefore cannot be expressed by using the same `StorageClass` of a `PersistentVolumeClaim`.

Volume snapshots provide Kubernetes users with a standardized way to copy a volume's contents at a particular point in time without creating an entirely new volume. This functionality enables, for example, database administrators to backup databases before performing edit or delete modifications.

Users need to be aware of the following when using this feature:

- API Objects `VolumeSnapshot`, `VolumeSnapshotContent`, and `VolumeSnapshotClass` are [CRDs](#), not part of the core API.

- *VolumeSnapshot support is only available for CSI drivers.*
- *As part of the deployment process of VolumeSnapshot, the Kubernetes team provides a snapshot controller to be deployed into the control plane, and a sidecar helper container called csi-snapshotter to be deployed together with the CSI driver. The snapshot controller watches VolumeSnapshot and VolumeSnapshotContent objects and is responsible for the creation and deletion of VolumeSnapshotContent object. The sidecar csi-snapshotter watches VolumeSnapshotContent objects and triggers CreateSnapshot and DeleteSnapshot operations against a CSI endpoint.*
- *There is also a validating webhook server which provides tightened validation on snapshot objects. This should be installed by the Kubernetes distros along with the snapshot controller and CRDs, not CSI drivers. It should be installed in all Kubernetes clusters that has the snapshot feature enabled.*
- *CSI drivers may or may not have implemented the volume snapshot functionality. The CSI drivers that have provided support for volume snapshot will likely use the csi-snapshotter. See [CSI Driver documentation](#) for details.*
- *The CRDs and snapshot controller installations are the responsibility of the Kubernetes distribution.*

Lifecycle of a volume snapshot and volume snapshot content

VolumeSnapshotContents are resources in the cluster. *VolumeSnapshots* are requests for those resources. The interaction between *VolumeSnapshotContents* and *VolumeSnapshots* follow this lifecycle:

Provisioning Volume Snapshot

There are two ways snapshots may be provisioned: pre-provisioned or dynamically provisioned.

Pre-provisioned

A cluster administrator creates a number of VolumeSnapshotContents. They carry the details of the real volume snapshot on the storage system which is available for use by cluster users. They exist in the Kubernetes API and are available for consumption.

Dynamic

Instead of using a pre-existing snapshot, you can request that a snapshot to be dynamically taken from a PersistentVolumeClaim. The [VolumeSnapshotClass](#) specifies storage provider-specific parameters to use when taking a snapshot.

Binding

The snapshot controller handles the binding of a `VolumeSnapshot` object with an appropriate `VolumeSnapshotContent` object, in both pre-provisioned and dynamically provisioned scenarios. The binding is a one-to-one mapping.

In the case of pre-provisioned binding, the `VolumeSnapshot` will remain unbound until the requested `VolumeSnapshotContent` object is created.

Persistent Volume Claim as Snapshot Source Protection

The purpose of this protection is to ensure that in-use [PersistentVolumeClaim](#) API objects are not removed from the system while a snapshot is being taken from it (as this may result in data loss).

While a snapshot is being taken of a `PersistentVolumeClaim`, that `PersistentVolumeClaim` is in-use. If you delete a `PersistentVolumeClaim` API object in active use as a snapshot source, the `PersistentVolumeClaim` object is not removed immediately. Instead, removal of the `PersistentVolumeClaim` object is postponed until the snapshot is `readyToUse` or aborted.

Delete

Deletion is triggered by deleting the `VolumeSnapshot` object, and the `DeletionPolicy` will be followed. If the `DeletionPolicy` is `Delete`, then the underlying storage snapshot will be deleted along with the `VolumeSnapshotContent` object. If the `DeletionPolicy` is `Retain`, then both the underlying snapshot and `VolumeSnapshotContent` remain.

VolumeSnapshots

Each `VolumeSnapshot` contains a `spec` and a `status`.

```
apiVersion: snapshot.storage.k8s.io/v1
kind: VolumeSnapshot
metadata:
  name: new-snapshot-test
spec:
  volumeSnapshotClassName: csi-hostpath-snapclass
  source:
    persistentVolumeClaimName: pvc-test
```

`persistentVolumeClaimName` is the name of the `PersistentVolumeClaim` data source for the snapshot. This field is required for dynamically provisioning a snapshot.

A volume snapshot can request a particular class by specifying the name of a [VolumeSnapshotClass](#) using the attribute `volumeSnapshotClassName`. If nothing is set, then the default class is used if available.

For pre-provisioned snapshots, you need to specify a `volumeSnapshotContentName` as the source for the snapshot as shown in the following example. The `volumeSnapshotContentName` source field is required for pre-provisioned snapshots.

```
apiVersion: snapshot.storage.k8s.io/v1
kind: VolumeSnapshot
metadata:
  name: test-snapshot
spec:
  source:
    volumeSnapshotContentName: test-content
```

Volume Snapshot Contents

Each `VolumeSnapshotContent` contains a `spec` and `status`. In dynamic provisioning, the `snapshot` common controller creates `VolumeSnapshotContent` objects. Here is an example:

```
apiVersion: snapshot.storage.k8s.io/v1
kind: VolumeSnapshotContent
metadata:
  name: snapcontent-72d9a349-aacd-42d2-a240-d775650d2455
spec:
  deletionPolicy: Delete
  driver: hostpath.csi.k8s.io
  source:
    volumeHandle: ee0cfb94-f8d4-11e9-b2d8-0242ac110002
    volumeSnapshotClassName: csi-hostpath-snapclass
    volumeSnapshotRef:
      name: new-snapshot-test
      namespace: default
      uid: 72d9a349-aacd-42d2-a240-d775650d2455
```

`volumeHandle` is the unique identifier of the volume created on the storage backend and returned by the CSI driver during the volume creation. This field is required for dynamically provisioning a snapshot. It specifies the volume source of the snapshot.

For pre-provisioned snapshots, you (as cluster administrator) are responsible for creating the `VolumeSnapshotContent` object as follows.

```
apiVersion: snapshot.storage.k8s.io/v1
kind: VolumeSnapshotContent
metadata:
  name: new-snapshot-content-test
spec:
  deletionPolicy: Delete
  driver: hostpath.csi.k8s.io
  source:
    snapshotHandle: 7bdd0de3-aaeb-11e8-9aae-0242ac110002
    volumeSnapshotRef:
```

```
name: new-snapshot-test
namespace: default
```

snapshotHandle is the unique identifier of the volume snapshot created on the storage backend. This field is required for the pre-provisioned snapshots. It specifies the CSI snapshot id on the storage system that this *VolumeSnapshotContent* represents.

Provisioning Volumes from Snapshots

You can provision a new volume, pre-populated with data from a snapshot, by using the *dataSource* field in the *PersistentVolumeClaim* object.

For more details, see [Volume Snapshot and Restore Volume from Snapshot](#).

Feedback

Was this page helpful?

Yes No

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

Last modified November 13, 2020 at 8:08 AM PST: [Add doc for snapshot GA \(#24849\) \(e62b6e1b1\)](#)

[Edit this page](#) [Create child page](#) [Create an issue](#)

- [Introduction](#)
- [Lifecycle of a volume snapshot and volume snapshot content](#)
 - [Provisioning Volume Snapshot](#)
 - [Binding](#)
 - [Persistent Volume Claim as Snapshot Source Protection](#)
 - [Delete](#)
- [VolumeSnapshots](#)
- [Volume Snapshot Contents](#)
- [Provisioning Volumes from Snapshots](#)

CSI Volume Cloning

This document describes the concept of cloning existing CSI Volumes in Kubernetes. Familiarity with [Volumes](#) is suggested.

Introduction

The [CSI](#) Volume Cloning feature adds support for specifying existing [PVCs](#) in the *dataSource* field to indicate a user would like to clone a [Volume](#).

A *Clone* is defined as a duplicate of an existing Kubernetes Volume that can be consumed as any standard Volume would be. The only difference is that upon provisioning, rather than creating a "new" empty Volume, the back end device creates an exact duplicate of the specified Volume.

The implementation of cloning, from the perspective of the Kubernetes API, simply adds the ability to specify an existing PVC as a *dataSource* during new PVC creation. The source PVC must be bound and available (not in use).

Users need to be aware of the following when using this feature:

- Cloning support (*VolumePVCDataSource*) is only available for CSI drivers.
- Cloning support is only available for dynamic provisioners.
- CSI drivers may or may not have implemented the volume cloning functionality.
- You can only clone a PVC when it exists in the same namespace as the destination PVC (source and destination must be in the same namespace).
- Cloning is only supported within the same Storage Class.
 - Destination volume must be the same storage class as the source
 - Default storage class can be used and *storageClassName* omitted in the spec
- Cloning can only be performed between two volumes that use the same *VolumeMode* setting (if you request a block mode volume, the source MUST also be block mode)

Provisioning

Clones are provisioned just like any other PVC with the exception of adding a *dataSource* that references an existing PVC in the same namespace.

```
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: clone-of-pvc-1
  namespace: myns
spec:
  accessModes:
  - ReadWriteOnce
  storageClassName: cloning
  resources:
    requests:
      storage: 5Gi
  dataSource:
    kind: PersistentVolumeClaim
    name: pvc-1
```

Note: You must specify a capacity value for *spec.resources.requests.storage*, and the value you specify must be the same or larger than the capacity of the source volume.

The result is a new PVC with the name `clone-of-pvc-1` that has the exact same content as the specified source `pvc-1`.

Usage

Upon availability of the new PVC, the cloned PVC is consumed the same as other PVC. It's also expected at this point that the newly created PVC is an independent object. It can be consumed, cloned, snapshotted, or deleted independently and without consideration for it's original dataSource PVC. This also implies that the source is not linked in any way to the newly created clone, it may also be modified or deleted without affecting the newly created clone.

Feedback

Was this page helpful?

Yes *No*

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

*Last modified May 30, 2020 at 3:10 PM PST: [add en pages \(ecc27bbbe\)](#)
[Edit this page](#) [Create child page](#) [Create an issue](#)*

- [Introduction](#)
- [Provisioning](#)
- [Usage](#)

Storage Classes

This document describes the concept of a StorageClass in Kubernetes. Familiarity with [volumes](#) and [persistent volumes](#) is suggested.

Introduction

A StorageClass provides a way for administrators to describe the "classes" of storage they offer. Different classes might map to quality-of-service levels, or to backup policies, or to arbitrary policies determined by the cluster administrators. Kubernetes itself is unopinionated about what classes represent. This concept is sometimes called "profiles" in other storage systems.

The StorageClass Resource

Each `StorageClass` contains the fields `provisioner`, `parameters`, and `reclaimPolicy`, which are used when a `PersistentVolume` belonging to the class needs to be dynamically provisioned.

The name of a `StorageClass` object is significant, and is how users can request a particular class. Administrators set the name and other parameters of a class when first creating `StorageClass` objects, and the objects cannot be updated once they are created.

Administrators can specify a default `StorageClass` just for PVCs that don't request any particular class to bind to: see the [PersistentVolumeClaim section](#) for details.

```
apiVersion: storage.k8s.io/v1
kind: StorageClass
metadata:
  name: standard
provisioner: kubernetes.io/aws-ebs
parameters:
  type: gp2
reclaimPolicy: Retain
allowVolumeExpansion: true
mountOptions:
  - debug
volumeBindingMode: Immediate
```

Provisioner

Each `StorageClass` has a provisioner that determines what volume plugin is used for provisioning PVs. This field must be specified.

Volume Plugin	Internal Provisioner	Config Example
AWElasticBlockStore	“	AWS EBS
AzureFile	“	Azure File
AzureDisk	“	Azure Disk
CephFS	-	-
Cinder	“	OpenStack Cinder
FC	-	-
FlexVolume	-	-
Flocker	“	-
GCEPersistentDisk	“	GCE PD
Glusterfs	“	Glusterfs
iSCSI	-	-
Quobyte	“	Quobyte
NFS	-	-
RBD	“	Ceph RBD
VsphereVolume	“	vSphere

Volume Plugin	Internal Provisioner	Config Example
PortworxVolume	“	Portworx Volume
ScaleIO	“	ScaleIO
StorageOS	“	StorageOS
Local	-	Local

You are not restricted to specifying the "internal" provisioners listed here (whose names are prefixed with "kubernetes.io" and shipped alongside Kubernetes). You can also run and specify external provisioners, which are independent programs that follow a [specification](#) defined by Kubernetes. Authors of external provisioners have full discretion over where their code lives, how the provisioner is shipped, how it needs to be run, what volume plugin it uses (including Flex), etc. The repository [kubernetes-sigs/sig-storage-lib-external-provisioner](#) houses a library for writing external provisioners that implements the bulk of the specification. Some external provisioners are listed under the repository [kubernetes-sigs/sig-storage-lib-external-provisioner](#).

For example, NFS doesn't provide an internal provisioner, but an external provisioner can be used. There are also cases when 3rd party storage vendors provide their own external provisioner.

Reclaim Policy

PersistentVolumes that are dynamically created by a StorageClass will have the reclaim policy specified in the `reclaimPolicy` field of the class, which can be either `Delete` or `Retain`. If no `reclaimPolicy` is specified when a StorageClass object is created, it will default to `Delete`.

PersistentVolumes that are created manually and managed via a StorageClass will have whatever reclaim policy they were assigned at creation.

Allow Volume Expansion

FEATURE STATE: Kubernetes v1.11 [beta]

PersistentVolumes can be configured to be expandable. This feature when set to `true`, allows the users to resize the volume by editing the corresponding PVC object.

The following types of volumes support volume expansion, when the underlying StorageClass has the field `allowVolumeExpansion` set to `true`.

Volume type	Required Kubernetes version
gcePersistentDisk	1.11
awsElasticBlockStore	1.11
Cinder	1.11
glusterfs	1.11
rbd	1.11

Volume type	Required Kubernetes version
Azure File	1.11
Azure Disk	1.11
Portworx	1.11
FlexVolume	1.13
CSI	1.14 (alpha), 1.16 (beta)

Note: You can only use the volume expansion feature to grow a Volume, not to shrink it.

Mount Options

PersistentVolumes that are dynamically created by a StorageClass will have the mount options specified in the `mountOptions` field of the class.

If the volume plugin does not support mount options but mount options are specified, provisioning will fail. Mount options are not validated on either the class or PV, so mount of the PV will simply fail if one is invalid.

Volume Binding Mode

The `volumeBindingMode` field controls when [volume binding and dynamic provisioning](#) should occur.

By default, the `Immediate` mode indicates that volume binding and dynamic provisioning occurs once the PersistentVolumeClaim is created. For storage backends that are topology-constrained and not globally accessible from all Nodes in the cluster, PersistentVolumes will be bound or provisioned without knowledge of the Pod's scheduling requirements. This may result in unschedulable Pods.

A cluster administrator can address this issue by specifying the `WaitForFirstConsumer` mode which will delay the binding and provisioning of a PersistentVolume until a Pod using the PersistentVolumeClaim is created. PersistentVolumes will be selected or provisioned conforming to the topology that is specified by the Pod's scheduling constraints. These include, but are not limited to, [resource requirements](#), [node selectors](#), [pod affinity and anti-affinity](#), and [taints and tolerations](#).

The following plugins support `WaitForFirstConsumer` with dynamic provisioning:

- [AWS EBS](#)
- [GCE Persistent Disk](#)
- [Azure Disk](#)

The following plugins support `WaitForFirstConsumer` with pre-created PersistentVolume binding:

- All of the above
- [Local](#)

FEATURE STATE: Kubernetes v1.17 [stable]

[CSI volumes](#) are also supported with dynamic provisioning and pre-created PVs, but you'll need to look at the documentation for a specific CSI driver to see its supported topology keys and examples.

Allowed Topologies

When a cluster operator specifies the `WaitForFirstConsumer` volume binding mode, it is no longer necessary to restrict provisioning to specific topologies in most situations. However, if still required, `allowedTopologies` can be specified.

This example demonstrates how to restrict the topology of provisioned volumes to specific zones and should be used as a replacement for the `zone` and `zones` parameters for the supported plugins.

```
apiVersion: storage.k8s.io/v1
kind: StorageClass
metadata:
  name: standard
provisioner: kubernetes.io/gce-pd
parameters:
  type: pd-standard
volumeBindingMode: WaitForFirstConsumer
allowedTopologies:
- matchLabelExpressions:
  - key: failure-domain.beta.kubernetes.io/zone
    values:
      - us-central1-a
      - us-central1-b
```

Parameters

Storage Classes have parameters that describe volumes belonging to the storage class. Different parameters may be accepted depending on the provisioner. For example, the value `iol`, for the parameter `type`, and the parameter `iopsPerGB` are specific to EBS. When a parameter is omitted, some default is used.

There can be at most 512 parameters defined for a StorageClass. The total length of the parameters object including its keys and values cannot exceed 256 KiB.

AWS EBS

```

apiVersion: storage.k8s.io/v1
kind: StorageClass
metadata:
  name: slow
provisioner: kubernetes.io/aws-ebs
parameters:
  type: io1
  iopsPerGB: "10"
  fsType: ext4

```

- **type**: *io1, gp2, sc1, st1*. See [AWS docs](#) for details. Default: *gp2*.
- **zone** (*Deprecated*): AWS zone. If neither **zone** nor **zones** is specified, volumes are generally round-robin-ed across all active zones where Kubernetes cluster has a node. **zone** and **zones** parameters must not be used at the same time.
- **zones** (*Deprecated*): A comma separated list of AWS zone(s). If neither **zone** nor **zones** is specified, volumes are generally round-robin-ed across all active zones where Kubernetes cluster has a node. **zone** and **zones** parameters must not be used at the same time.
- **iopsPerGB**: only for *io1* volumes. I/O operations per second per GiB. AWS volume plugin multiplies this with size of requested volume to compute IOPS of the volume and caps it at 20 000 IOPS (maximum supported by AWS, see [AWS docs](#)). A string is expected here, i.e. "*10*", not *10*.
- **fsType**: *fsType* that is supported by kubernetes. Default: "*ext4*".
- **encrypted**: denotes whether the EBS volume should be encrypted or not. Valid values are "*true*" or "*false*". A string is expected here, i.e. "*true*", not *true*.
- **kmsKeyId**: optional. The full Amazon Resource Name of the key to use when encrypting the volume. If none is supplied but **encrypted** is true, a key is generated by AWS. See AWS docs for valid ARN value.

Note: **zone** and **zones** parameters are deprecated and replaced with [allowedTopologies](#)

GCE PD

```

apiVersion: storage.k8s.io/v1
kind: StorageClass
metadata:
  name: slow
provisioner: kubernetes.io/gce-pd
parameters:
  type: pd-standard
  fstype: ext4
  replication-type: none

```

- **type**: *pd-standard* or *pd-ssd*. Default: *pd-standard*

- *zone* (Deprecated): GCE zone. If neither *zone* nor *zones* is specified, volumes are generally round-robin-ed across all active zones where Kubernetes cluster has a node. *zone* and *zones* parameters must not be used at the same time.
- *zones* (Deprecated): A comma separated list of GCE zone(s). If neither *zone* nor *zones* is specified, volumes are generally round-robin-ed across all active zones where Kubernetes cluster has a node. *zone* and *zones* parameters must not be used at the same time.
- *fstype*: ext4 or xfs. Default: ext4. The defined filesystem type must be supported by the host operating system.
- *replication-type*: none or regional-pd. Default: none.

If *replication-type* is set to *none*, a regular (zonal) PD will be provisioned.

If *replication-type* is set to *regional-pd*, a [Regional Persistent Disk](#) will be provisioned. It's highly recommended to have *volumeBindingMode*: *WaitForFirstConsumer* set, in which case when you create a Pod that consumes a *PersistentVolumeClaim* which uses this *StorageClass*, a Regional Persistent Disk is provisioned with two zones. One zone is the same as the zone that the Pod is scheduled in. The other zone is randomly picked from the zones available to the cluster. Disk zones can be further constrained using *allowedTopologies*.

Note: *zone* and *zones* parameters are deprecated and replaced with [allowedTopologies](#)

Glusterfs

```
apiVersion: storage.k8s.io/v1
kind: StorageClass
metadata:
  name: slow
provisioner: kubernetes.io/glusterfs
parameters:
  resturl: "http://127.0.0.1:8081"
  clusterid: "630372ccdc720a92c681fb928f27b53f"
  restauthenabled: "true"
  restuser: "admin"
  secretNamespace: "default"
  secretName: "heketi-secret"
  gidMin: "40000"
  gidMax: "50000"
  volumetype: "replicate:3"
```

- *resturl*: Gluster REST service/Heketi service url which provision gluster volumes on demand. The general format should be `IPaddress:Port` and this is a mandatory parameter for GlusterFS dynamic provisioner. If Heketi service is exposed as a routable service in openshift/kubernetes setup, this can have a format similar to `http://heketi-storage-project.cloudapps.mystorage.com` where the fqdn is a resolvable Heketi service url.
- *restauthenabled* : Gluster REST service authentication boolean that enables authentication to the REST server. If this value is "true", *restuser* and *restuserkey* or *secretNamespace + secretName* have to be filled. This option is deprecated, authentication is enabled when any of *restuser*, *restuserkey*, *secretName* or *secretNamespace* is specified.
- *restuser* : Gluster REST service/Heketi user who has access to create volumes in the Gluster Trusted Pool.
- *restuserkey* : Gluster REST service/Heketi user's password which will be used for authentication to the REST server. This parameter is deprecated in favor of *secretNamespace + secretName*.
- *secretNamespace*, *secretName* : Identification of Secret instance that contains user password to use when talking to Gluster REST service. These parameters are optional, empty password will be used when both *secretNamespace* and *secretName* are omitted. The provided secret must have type "`kubernetes.io/glusterfs`", for example created in this way:

```
kubectl create secret generic heketi-secret \
--type="kubernetes.io/glusterfs" --from-
literal=key='opensesame' \
--namespace=default
```

Example of a secret can be found in [glusterfs-provisioning-secret.yaml](#).

- *clusterid*: `630372ccdc720a92c681fb928f27b53f` is the ID of the cluster which will be used by Heketi when provisioning the volume. It can also be a list of clusterids, for example: `"8452344e2becec931ece4e33c4674e4e,42982310de6c63381718ccfa6d8cf397"`. This is an optional parameter.
- *gidMin*, *gidMax* : The minimum and maximum value of GID range for the StorageClass. A unique value (GID) in this range (*gidMin-gidMax*) will be used for dynamically provisioned volumes. These are optional values. If not specified, the volume will be provisioned with a value between 2000-2147483647 which are defaults for *gidMin* and *gidMax* respectively.
- *volumetype* : The volume type and its parameters can be configured with this optional value. If the volume type is not mentioned, it's up to the provisioner to decide the volume type.

For example:

- Replica volume: `volumetype: replicate:3` where '3' is replica count.
- Disperse/EC volume: `volumetype: disperse:4:2` where '4' is data and '2' is the redundancy count.
- Distribute volume: `volumetype: none`

For available volume types and administration options, refer to the [Administration Guide](#).

For further reference information, see [How to configure Heketi](#).

When persistent volumes are dynamically provisioned, the Gluster plugin automatically creates an endpoint and a headless service in the name `gluster-dynamic-<claimname>`. The dynamic endpoint and service are automatically deleted when the persistent volume claim is deleted.

OpenStack Cinder

```
apiVersion: storage.k8s.io/v1
kind: StorageClass
metadata:
  name: gold
provisioner: kubernetes.io/cinder
parameters:
  availability: nova
```

- **availability:** Availability Zone. If not specified, volumes are generally round-robin-ed across all active zones where Kubernetes cluster has a node.

Note:

FEATURE STATE: Kubernetes v1.11 [deprecated]

This internal provisioner of OpenStack is deprecated. Please use [the external cloud provider for OpenStack](#).

vSphere

There are two types of provisioners for vSphere storage classes:

- [CSI provisioner](#): `csi.vsphere.vmware.com`
- [vCP provisioner](#): `kubernetes.io/vsphere-volume`

In-tree provisioners are [deprecated](#). For more information on the CSI provisioner, see [Kubernetes vSphere CSI Driver](#) and [vSphereVolume CSI migration](#).

CSI Provisioner

The vSphere CSI StorageClass provisioner works with Tanzu Kubernetes clusters. For an example, refer to the [vSphere CSI repository](#).

vCP Provisioner

The following examples use the VMware Cloud Provider (vCP) StorageClass provisioner.

1. Create a StorageClass with a user specified disk format.

```
apiVersion: storage.k8s.io/v1
kind: StorageClass
metadata:
  name: fast
provisioner: kubernetes.io/vsphere-volume
parameters:
  diskformat: zeroedthick
```

diskformat: thin, zeroedthick and eagerzeroedthick. Default: "thin".

2. Create a StorageClass with a disk format on a user specified datastore.

```
apiVersion: storage.k8s.io/v1
kind: StorageClass
metadata:
  name: fast
provisioner: kubernetes.io/vsphere-volume
parameters:
  diskformat: zeroedthick
  datastore: VSANDatastore
```

datastore: The user can also specify the datastore in the StorageClass. The volume will be created on the datastore specified in the StorageClass, which in this case is VSANDatastore. This field is optional. If the datastore is not specified, then the volume will be created on the datastore specified in the vSphere config file used to initialize the vSphere Cloud Provider.

3. Storage Policy Management inside kubernetes

- Using existing vCenter SPBM policy

One of the most important features of vSphere for Storage Management is policy based Management. Storage Policy Based Management (SPBM) is a storage policy framework that provides a single unified control plane across a broad range of data services and storage solutions. SPBM enables vSphere administrators to overcome upfront storage provisioning challenges, such as capacity planning, differentiated service levels and managing capacity headroom.

The SPBM policies can be specified in the StorageClass using the storagePolicyName parameter.

- *Virtual SAN policy support inside Kubernetes*

Vsphere Infrastructure (VI) Admins will have the ability to specify custom Virtual SAN Storage Capabilities during dynamic volume provisioning. You can now define storage requirements, such as performance and availability, in the form of storage capabilities during dynamic volume provisioning. The storage capability requirements are converted into a Virtual SAN policy which are then pushed down to the Virtual SAN layer when a persistent volume (virtual disk) is being created. The virtual disk is distributed across the Virtual SAN datastore to meet the requirements.

You can see [Storage Policy Based Management for dynamic provisioning of volumes](#) for more details on how to use storage policies for persistent volumes management.

There are few [vSphere examples](#) which you try out for persistent volume management inside Kubernetes for vSphere.

Ceph RBD

```
apiVersion: storage.k8s.io/v1
kind: StorageClass
metadata:
  name: fast
provisioner: kubernetes.io/rbd
parameters:
  monitors: 10.16.153.105:6789
  adminId: kube
  adminSecretName: ceph-secret
  adminSecretNamespace: kube-system
  pool: kube
  userId: kube
  userSecretName: ceph-secret-user
  userSecretNamespace: default
  fsType: ext4
```

```
imageFormat: "2"
imageFeatures: "layering"
```

- *monitors*: Ceph monitors, comma delimited. This parameter is required.
- *adminId*: Ceph client ID that is capable of creating images in the pool. Default is "admin".
- *adminSecretName*: Secret Name for *adminId*. This parameter is required. The provided secret must have type "kubernetes.io/rbd".
- *adminSecretNamespace*: The namespace for *adminSecretName*. Default is "default".
- *pool*: Ceph RBD pool. Default is "rbd".
- *userId*: Ceph client ID that is used to map the RBD image. Default is the same as *adminId*.
- *userSecretName*: The name of Ceph Secret for *userId* to map RBD image. It must exist in the same namespace as PVCs. This parameter is required. The provided secret must have type "kubernetes.io/rbd", for example created in this way:

```
kubectl create secret generic ceph-secret --type="kubernetes.io/rbd" \
--from-literal=key='QVFEQ1pMdFhPUUnQrSmhBQUFYaERWNHJsZ3BsMmN
jcDR6RFZST0E9PQ==' \
--namespace=kube-system
```

- *userSecretNamespace*: The namespace for *userSecretName*.
- *fsType*: fsType that is supported by kubernetes. Default: "ext4".
- *imageFormat*: Ceph RBD image format, "1" or "2". Default is "2".
- *imageFeatures*: This parameter is optional and should only be used if you set *imageFormat* to "2". Currently supported features are *layering* only. Default is "", and no features are turned on.

Quobyte

```
apiVersion: storage.k8s.io/v1
kind: StorageClass
metadata:
  name: slow
provisioner: kubernetes.io/quobyte
parameters:
  quobyteAPIServer: "http://138.68.74.142:7860"
  registry: "138.68.74.142:7861"
```

```

adminSecretName: "quobyte-admin-secret"
adminSecretNamespace: "kube-system"
user: "root"
group: "root"
quobyteConfig: "BASE"
quobyteTenant: "DEFAULT"

```

- **quobyteAPIServer**: API Server of Quobyte in the format "`http(s)://api-server:7860`"
- **registry**: Quobyte registry to use to mount the volume. You can specify the registry as `<host>:<port>` pair or if you want to specify multiple registries you just have to put a comma between them e.g. `<host1>:<port>,<host2>:<port>,<host3>:<port>`. The host can be an IP address or if you have a working DNS you can also provide the DNS names.
- **adminSecretNamespace**: The namespace for **adminSecretName**. Default is "default".
- **adminSecretName**: secret that holds information about the Quobyte user and the password to authenticate against the API server. The provided secret must have type "kubernetes.io/quobyte" and the keys `user` and `password`, for example:

```

kubectl create secret generic quobyte-admin-secret \
  --type="kubernetes.io/quobyte" --from-literal=user='admin' \
  --from-literal=password='opensesame' \
  --namespace=kube-system

```

- **user**: maps all access to this user. Default is "root".
- **group**: maps all access to this group. Default is "nfsnobody".
- **quobyteConfig**: use the specified configuration to create the volume. You can create a new configuration or modify an existing one with the Web console or the quobyte CLI. Default is "BASE".
- **quobyteTenant**: use the specified tenant ID to create/delete the volume. This Quobyte tenant has to be already present in Quobyte. Default is "DEFAULT".

Azure Disk

Azure Unmanaged Disk storage class

```

apiVersion: storage.k8s.io/v1
kind: StorageClass
metadata:
  name: slow

```

```
provisioner: kubernetes.io/azure-disk
parameters:
  skuName: Standard_LRS
  location: eastus
  storageAccount: azure_storage_account_name
```

- **skuName**: Azure storage account Sku tier. Default is empty.
- **location**: Azure storage account location. Default is empty.
- **storageAccount**: Azure storage account name. If a storage account is provided, it must reside in the same resource group as the cluster, and location is ignored. If a storage account is not provided, a new storage account will be created in the same resource group as the cluster.

Azure Disk storage class (starting from v1.7.2)

```
apiVersion: storage.k8s.io/v1
kind: StorageClass
metadata:
  name: slow
provisioner: kubernetes.io/azure-disk
parameters:
  storageaccounttype: Standard_LRS
  kind: Shared
```

- **storageaccounttype**: Azure storage account Sku tier. Default is empty.
 - **kind**: Possible values are *shared* (default), *dedicated*, and *managed*. When **kind** is *shared*, all unmanaged disks are created in a few shared storage accounts in the same resource group as the cluster. When **kind** is *dedicated*, a new dedicated storage account will be created for the new unmanaged disk in the same resource group as the cluster. When **kind** is *managed*, all managed disks are created in the same resource group as the cluster.
 - **resourceGroup**: Specify the resource group in which the Azure disk will be created. It must be an existing resource group name. If it is unspecified, the disk will be placed in the same resource group as the current Kubernetes cluster.
-
- Premium VM can attach both *Standard_LRS* and *Premium_LRS* disks, while Standard VM can only attach *Standard_LRS* disks.
 - Managed VM can only attach managed disks and unmanaged VM can only attach unmanaged disks.

Azure File

```
apiVersion: storage.k8s.io/v1
kind: StorageClass
```

```

metadata:
  name: azurefile
provisioner: kubernetes.io/azure-file
parameters:
  skuName: Standard_LRS
  location: eastus
  storageAccount: azure_storage_account_name

```

- **skuName:** Azure storage account Sku tier. Default is empty.
- **location:** Azure storage account location. Default is empty.
- **storageAccount:** Azure storage account name. Default is empty. If a storage account is not provided, all storage accounts associated with the resource group are searched to find one that matches **skuName** and **location**. If a storage account is provided, it must reside in the same resource group as the cluster, and **skuName** and **location** are ignored.
- **secretNamespace:** the namespace of the secret that contains the Azure Storage Account Name and Key. Default is the same as the Pod.
- **secretName:** the name of the secret that contains the Azure Storage Account Name and Key. Default is `azure-storage-account-<accountName>-secret`
- **readOnly:** a flag indicating whether the storage will be mounted as read only. Defaults to false which means a read/write mount. This setting will impact the `ReadOnly` setting in `VolumeMounts` as well.

During storage provisioning, a secret named by `secretName` is created for the mounting credentials. If the cluster has enabled both [RBAC](#) and [Controller Roles](#), add the `create` permission of resource secret for clusterrole `system:controller:persistent-volume-binder`.

In a multi-tenancy context, it is strongly recommended to set the value for `secretNamespace` explicitly, otherwise the storage account credentials may be read by other users.

Portworx Volume

```

apiVersion: storage.k8s.io/v1
kind: StorageClass
metadata:
  name: portworx-io-priority-high
provisioner: kubernetes.io/portworx-volume
parameters:
  repl: "1"
  snap_interval: "70"
  priority_io: "high"

```

- **fs:** filesystem to be laid out: `none/xfs/ext4` (default: `ext4`).

- *block_size*: block size in Kbytes (default: 32).
- *repl*: number of synchronous replicas to be provided in the form of replication factor 1..3 (default: 1) A string is expected here i.e. "1" and not 1.
- *priority_io*: determines whether the volume will be created from higher performance or a lower priority storage high/medium/low (default: low).
- *snap_interval*: clock/time interval in minutes for when to trigger snapshots. Snapshots are incremental based on difference with the prior snapshot, 0 disables snaps (default: 0). A string is expected here i.e. "70" and not 70.
- *aggregation_level*: specifies the number of chunks the volume would be distributed into, 0 indicates a non-aggregated volume (default: 0). A string is expected here i.e. "0" and not 0
- *ephemeral*: specifies whether the volume should be cleaned-up after unmount or should be persistent. *emptyDir* use case can set this value to true and persistent volumes use case such as for databases like Cassandra should set to false, true/false (default false). A string is expected here i.e. "true" and not true.

ScaleIO

```
apiVersion: storage.k8s.io/v1
kind: StorageClass
metadata:
  name: slow
provisioner: kubernetes.io/scaleio
parameters:
  gateway: https://192.168.99.200:443/api
  system: scaleio
  protectionDomain: pd0
  storagePool: sp1
  storageMode: ThinProvisioned
  secretRef: sio-secret
  readOnly: false
  fsType: xfs
```

- *provisioner*: attribute is set to kubernetes.io/scaleio
- *gateway*: address to a ScaleIO API gateway (required)
- *system*: the name of the ScaleIO system (required)
- *protectionDomain*: the name of the ScaleIO protection domain (required)
- *storagePool*: the name of the volume storage pool (required)
- *storageMode*: the storage provision mode: ThinProvisioned (default) or ThickProvisioned
- *secretRef*: reference to a configured Secret object (required)
- *readOnly*: specifies the access mode to the mounted volume (default false)
- *fsType*: the file system to use for the volume (default ext4)

The ScaleIO Kubernetes volume plugin requires a configured Secret object. The secret must be created with type `kubernetes.io/scaleio` and use the same namespace value as that of the PVC where it is referenced as shown in the following command:

```
kubectl create secret generic sio-secret --type="kubernetes.io/scaleio" \
--from-literal=username=sioadmin --from-literal=password=d2NABDNjMA== \
--namespace=default
```

StorageOS

```
apiVersion: storage.k8s.io/v1
kind: StorageClass
metadata:
  name: fast
provisioner: kubernetes.io/storageos
parameters:
  pool: default
  description: Kubernetes volume
  fsType: ext4
  adminSecretNamespace: default
  adminSecretName: storageos-secret
```

- **pool:** The name of the StorageOS distributed capacity pool to provision the volume from. Uses the `default` pool which is normally present if not specified.
- **description:** The description to assign to volumes that were created dynamically. All volume descriptions will be the same for the storage class, but different storage classes can be used to allow descriptions for different use cases. Defaults to `Kubernetes volume`.
- **fsType:** The default filesystem type to request. Note that user-defined rules within StorageOS may override this value. Defaults to `ext4`.
- **adminSecretNamespace:** The namespace where the API configuration secret is located. Required if `adminSecretName` set.
- **adminSecretName:** The name of the secret to use for obtaining the StorageOS API credentials. If not specified, default values will be attempted.

The StorageOS Kubernetes volume plugin can use a Secret object to specify an endpoint and credentials to access the StorageOS API. This is only required when the defaults have been changed. The secret must be created with type `kubernetes.io/storageos` as shown in the following command:

```
kubectl create secret generic storageos-secret \
--type="kubernetes.io/storageos" \
--from-literal=apiAddress=tcp://localhost:5705 \
--from-literal=apiUsername=storageos \
--from-literal=apiPassword=storageos \
--namespace=default
```

Secrets used for dynamically provisioned volumes may be created in any namespace and referenced with the `adminSecretNamespace` parameter. Secrets used by pre-provisioned volumes must be created in the same namespace as the PVC that references it.

Local

FEATURE STATE: Kubernetes v1.14 [stable]

```
apiVersion: storage.k8s.io/v1
kind: StorageClass
metadata:
  name: local-storage
provisioner: kubernetes.io/no-provisioner
volumeBindingMode: WaitForFirstConsumer
```

Local volumes do not currently support dynamic provisioning, however a StorageClass should still be created to delay volume binding until Pod scheduling. This is specified by the `WaitForFirstConsumer` volume binding mode.

Delaying volume binding allows the scheduler to consider all of a Pod's scheduling constraints when choosing an appropriate PersistentVolume for a PersistentVolumeClaim.

Feedback

Was this page helpful?

Yes No

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

Last modified November 11, 2020 at 2:54 PM PST: [Move link away from deprecated external-storage repo \(e3db38188\)](#)

[Edit this page](#) [Create child page](#) [Create an issue](#)

- [Introduction](#)

- [The StorageClass Resource](#)
 - [Provisioner](#)
 - [Reclaim Policy](#)
 - [Allow Volume Expansion](#)
 - [Mount Options](#)
 - [Volume Binding Mode](#)
 - [Allowed Topologies](#)
- [Parameters](#)
 - [AWS EBS](#)
 - [GCE PD](#)
 - [Glusterfs](#)
 - [OpenStack Cinder](#)
 - [vSphere](#)
 - [Ceph RBD](#)
 - [Quobyte](#)
 - [Azure Disk](#)
 - [Azure File](#)
 - [Portworx Volume](#)
 - [ScaleIO](#)
 - [StorageOS](#)
 - [Local](#)

Volume Snapshot Classes

This document describes the concept of `VolumeSnapshotClass` in Kubernetes. Familiarity with [volume snapshots](#) and [storage classes](#) is suggested.

Introduction

Just like `StorageClass` provides a way for administrators to describe the "classes" of storage they offer when provisioning a volume, `VolumeSnapshotClass` provides a way to describe the "classes" of storage when provisioning a volume snapshot.

The `VolumeSnapshotClass` Resource

Each `VolumeSnapshotClass` contains the fields `driver`, `deletionPolicy`, and `parameters`, which are used when a `VolumeSnapshot` belonging to the class needs to be dynamically provisioned.

The name of a `VolumeSnapshotClass` object is significant, and is how users can request a particular class. Administrators set the name and other parameters of a class when first creating `VolumeSnapshotClass` objects, and the objects cannot be updated once they are created.

```
apiVersion: snapshot.storage.k8s.io/v1
kind: VolumeSnapshotClass
metadata:
```

```
  name: csi-hostpath-snapclass
  driver: hostpath.csi.k8s.io
  deletionPolicy: Delete
  parameters:
```

Administrators can specify a default VolumeSnapshotClass for VolumeSnapshots that don't request any particular class to bind to by adding the `snapshot.storage.kubernetes.io/is-default-class: "true"` annotation:

```
apiVersion: snapshot.storage.k8s.io/v1
kind: VolumeSnapshotClass
metadata:
  name: csi-hostpath-snapclass
  annotations:
    snapshot.storage.kubernetes.io/is-default-class: "true"
  driver: hostpath.csi.k8s.io
  deletionPolicy: Delete
  parameters:
```

Driver

Volume snapshot classes have a `driver` that determines what CSI volume plugin is used for provisioning VolumeSnapshots. This field must be specified.

DeletionPolicy

Volume snapshot classes have a `deletionPolicy`. It enables you to configure what happens to a `VolumeSnapshotContent` when the `VolumeSnapshot` object it is bound to is to be deleted. The `deletionPolicy` of a volume snapshot can either be `Retain` or `Delete`. This field must be specified.

If the `deletionPolicy` is `Delete`, then the underlying storage snapshot will be deleted along with the `VolumeSnapshotContent` object. If the `deletionPolicy` is `Retain`, then both the underlying snapshot and `VolumeSnapshotContent` remain.

Parameters

Volume snapshot classes have parameters that describe volume snapshots belonging to the volume snapshot class. Different parameters may be accepted depending on the `driver`.

Feedback

Was this page helpful?

Yes No

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

Last modified November 13, 2020 at 8:08 AM PST: [Add doc for snapshot GA \(#24849\) \(e62b6e1b1\)](#)

[Edit this page](#) [Create child page](#) [Create an issue](#)

- [Introduction](#)
- [The VolumeSnapshotClass Resource](#)
 - [Driver](#)
 - [DeletionPolicy](#)
- [Parameters](#)

Dynamic Volume Provisioning

Dynamic volume provisioning allows storage volumes to be created on-demand. Without dynamic provisioning, cluster administrators have to manually make calls to their cloud or storage provider to create new storage volumes, and then create [PersistentVolume objects](#) to represent them in Kubernetes. The dynamic provisioning feature eliminates the need for cluster administrators to pre-provision storage. Instead, it automatically provisions storage when it is requested by users.

Background

The implementation of dynamic volume provisioning is based on the API object `StorageClass` from the API group `storage.k8s.io`. A cluster administrator can define as many `StorageClass` objects as needed, each specifying a volume plugin (aka provisioner) that provisions a volume and the set of parameters to pass to that provisioner when provisioning. A cluster administrator can define and expose multiple flavors of storage (from the same or different storage systems) within a cluster, each with a custom set of parameters. This design also ensures that end users don't have to worry about the complexity and nuances of how storage is provisioned, but still have the ability to select from multiple storage options.

More information on storage classes can be found [here](#).

Enabling Dynamic Provisioning

To enable dynamic provisioning, a cluster administrator needs to pre-create one or more `StorageClass` objects for users. `StorageClass` objects define which provisioner should be used and what parameters should be passed to that provisioner when dynamic provisioning is invoked. The name of a `StorageClass` object must be a valid [DNS subdomain name](#).

The following manifest creates a storage class "slow" which provisions standard disk-like persistent disks.

```
apiVersion: storage.k8s.io/v1
kind: StorageClass
metadata:
  name: slow
provisioner: kubernetes.io/gce-pd
parameters:
  type: pd-standard
```

The following manifest creates a storage class "fast" which provisions SSD-like persistent disks.

```
apiVersion: storage.k8s.io/v1
kind: StorageClass
metadata:
  name: fast
provisioner: kubernetes.io/gce-pd
parameters:
  type: pd-ssd
```

Using Dynamic Provisioning

Users request dynamically provisioned storage by including a storage class in their `PersistentVolumeClaim`. Before Kubernetes v1.6, this was done via the `volume.beta.kubernetes.io/storage-class` annotation. However, this annotation is deprecated since v1.6. Users now can and should instead use the `storageClassName` field of the `PersistentVolumeClaim` object. The value of this field must match the name of a `StorageClass` configured by the administrator (see [below](#)).

To select the "fast" storage class, for example, a user would create the following `PersistentVolumeClaim`:

```
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: claim1
spec:
  accessModes:
    - ReadWriteOnce
  storageClassName: fast
  resources:
    requests:
      storage: 30Gi
```

This claim results in an SSD-like Persistent Disk being automatically provisioned. When the claim is deleted, the volume is destroyed.

Defaulting Behavior

Dynamic provisioning can be enabled on a cluster such that all claims are dynamically provisioned if no storage class is specified. A cluster administrator can enable this behavior by:

- *Marking one StorageClass object as default;*
- *Making sure that the [DefaultStorageClass admission controller](#) is enabled on the API server.*

An administrator can mark a specific StorageClass as default by adding the storageclass.kubernetes.io/is-default-class annotation to it. When a default StorageClass exists in a cluster and a user creates a PersistentVolumeClaim with storageClassName unspecified, the DefaultStorageClass admission controller automatically adds the storageClassName field pointing to the default storage class.

Note that there can be at most one default storage class on a cluster, or a PersistentVolumeClaim without storageClassName explicitly specified cannot be created.

Topology Awareness

In [Multi-Zone](#) clusters, Pods can be spread across Zones in a Region. Single-Zone storage backends should be provisioned in the Zones where Pods are scheduled. This can be accomplished by setting the [Volume Binding Mode](#).

Feedback

Was this page helpful?

Yes No

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

Last modified August 05, 2020 at 3:17 AM PST: [Replace special quote characters with normal ones. \(c6a96128c\)](#)

[Edit this page](#) [Create child page](#) [Create an issue](#)

- [Background](#)
- [Enabling Dynamic Provisioning](#)
- [Using Dynamic Provisioning](#)
- [Defaulting Behavior](#)
- [Topology Awareness](#)

Storage Capacity

Storage capacity is limited and may vary depending on the node on which a pod runs: network-attached storage might not be accessible by all nodes, or storage is local to a node to begin with.

FEATURE STATE: Kubernetes v1.19 [alpha]

This page describes how Kubernetes keeps track of storage capacity and how the scheduler uses that information to schedule Pods onto nodes that have access to enough storage capacity for the remaining missing volumes. Without storage capacity tracking, the scheduler may choose a node that doesn't have enough capacity to provision a volume and multiple scheduling retries will be needed.

Tracking storage capacity is supported for [Container Storage Interface \(CSI\)](#) drivers and [needs to be enabled](#) when installing a CSI driver.

API

There are two API extensions for this feature:

- *CSIStrorageCapacity objects: these get produced by a CSI driver in the namespace where the driver is installed. Each object contains capacity information for one storage class and defines which nodes have access to that storage.*
- *[The CSIDriverSpec.StorageCapacity field](#): when set to true, the Kubernetes scheduler will consider storage capacity for volumes that use the CSI driver.*

Scheduling

Storage capacity information is used by the Kubernetes scheduler if:

- *the CSIStrorageCapacity feature gate is true,*
- *a Pod uses a volume that has not been created yet,*
- *that volume uses a [StorageClass](#) which references a CSI driver and uses [WaitForFirstConsumer volume binding mode](#), and*
- *the CSIDriver object for the driver has StorageCapacity set to true.*

In that case, the scheduler only considers nodes for the Pod which have enough storage available to them. This check is very simplistic and only compares the size of the volume against the capacity listed in CSIStrorageCapacity objects with a topology that includes the node.

For volumes with [Immediate volume binding mode](#), the storage driver decides where to create the volume, independently of Pods that will use the volume. The scheduler then schedules Pods onto nodes where the volume is available after the volume has been created.

For [CSI ephemeral volumes](#), scheduling always happens without considering storage capacity. This is based on the assumption that this volume type is only used by special CSI drivers which are local to a node and do not need significant resources there.

Rescheduling

When a node has been selected for a Pod with `WaitForFirstConsumer` volumes, that decision is still tentative. The next step is that the CSI storage driver gets asked to create the volume with a hint that the volume is supposed to be available on the selected node.

Because Kubernetes might have chosen a node based on out-dated capacity information, it is possible that the volume cannot really be created. The node selection is then reset and the Kubernetes scheduler tries again to find a node for the Pod.

Limitations

Storage capacity tracking increases the chance that scheduling works on the first try, but cannot guarantee this because the scheduler has to decide based on potentially out-dated information. Usually, the same retry mechanism as for scheduling without any storage capacity information handles scheduling failures.

One situation where scheduling can fail permanently is when a Pod uses multiple volumes: one volume might have been created already in a topology segment which then does not have enough capacity left for another volume. Manual intervention is necessary to recover from this, for example by increasing capacity or deleting the volume that was already created. [Further work](#) is needed to handle this automatically.

Enabling storage capacity tracking

Storage capacity tracking is an alpha feature and only enabled when the CSI `StorageCapacity` [feature gate](#) and the `storage.k8s.io/v1alpha1 API group` are enabled. For details on that, see the `--feature-gates` and `--runtime-config kube-apiserver parameters`.

A quick check whether a Kubernetes cluster supports the feature is to list `CSIStorageCapacity` objects with:

```
kubectl get csistoragecapacities --all-namespaces
```

If your cluster supports `CSIStorageCapacity`, the response is either a list of `CSIStorageCapacity` objects or:

```
No resources found
```

If not supported, this error is printed instead:

```
error: the server doesn't have a resource type  
"csistoragecapacities"
```

In addition to enabling the feature in the cluster, a CSI driver also has to support it. Please refer to the driver's documentation for details.

What's next

- For more information on the design, see the [Storage Capacity Constraints for Pod Scheduling KEP](#).
- For more information on further development of this feature, see the [enhancement tracking issue #1472](#).
- Learn about [Kubernetes Scheduler](#)

Feedback

Was this page helpful?

Yes No

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

Last modified December 13, 2020 at 8:24 PM PST: [Remove broken link to CSIStrageCapacity in API reference \(04664781a\)](#)
[Edit this page](#) [Create child page](#) [Create an issue](#)

- [API](#)
- [Scheduling](#)
- [Rescheduling](#)
- [Limitations](#)
- [Enabling storage capacity tracking](#)
- [What's next](#)

Ephemeral Volumes

This document describes ephemeral volumes in Kubernetes. Familiarity with [volumes](#) is suggested, in particular `PersistentVolumeClaim` and `PersistentVolume`.

Some application need additional storage but don't care whether that data is stored persistently across restarts. For example, caching services are often limited by memory size and can move infrequently used data into storage that is slower than memory with little impact on overall performance.

Other applications expect some read-only input data to be present in files, like configuration data or secret keys.

Ephemeral volumes are designed for these use cases. Because volumes follow the Pod's lifetime and get created and deleted along with the Pod, Pods can be stopped and restarted without being limited to where some persistent volume is available.

Ephemeral volumes are specified inline in the Pod spec, which simplifies application deployment and management.

Types of ephemeral volumes

Kubernetes supports several different kinds of ephemeral volumes for different purposes:

- [emptyDir](#): empty at Pod startup, with storage coming locally from the kubelet base directory (usually the root disk) or RAM
- [configMap](#), [downwardAPI](#), [secret](#): inject different kinds of Kubernetes data into a Pod
- [CSI ephemeral volumes](#): similar to the previous volume kinds, but provided by special [CSI drivers](#) which specifically [support this feature](#)
- [generic ephemeral volumes](#), which can be provided by all storage drivers that also support persistent volumes

[emptyDir](#), [configMap](#), [downwardAPI](#), [secret](#) are provided as [local ephemeral storage](#). They are managed by kubelet on each node.

CSI ephemeral volumes must be provided by third-party CSI storage drivers.

Generic ephemeral volumes can be provided by third-party CSI storage drivers, but also by any other storage driver that supports dynamic provisioning. Some CSI drivers are written specifically for CSI ephemeral volumes and do not support dynamic provisioning: those then cannot be used for generic ephemeral volumes.

The advantage of using third-party drivers is that they can offer functionality that Kubernetes itself does not support, for example storage with different performance characteristics than the disk that is managed by kubelet, or injecting different data.

CSI ephemeral volumes

FEATURE STATE: Kubernetes v1.16 [beta]

This feature requires the [CSIInlineVolume](#) [feature gate](#) to be enabled. It is enabled by default starting with Kubernetes 1.16.

Note: CSI ephemeral volumes are only supported by a subset of CSI drivers. The Kubernetes [CSI Drivers list](#) shows which drivers support ephemeral volumes.

Conceptually, CSI ephemeral volumes are similar to [configMap](#), [downwardAPI](#) and [secret](#) volume types: the storage is managed locally on each node and is created together with other local resources after a Pod has been scheduled onto a node. Kubernetes has no concept of rescheduling Pods

anymore at this stage. Volume creation has to be unlikely to fail, otherwise Pod startup gets stuck. In particular, [storage capacity aware Pod scheduling](#) is not supported for these volumes. They are currently also not covered by the storage resource usage limits of a Pod, because that is something that kubelet can only enforce for storage that it manages itself.

Here's an example manifest for a Pod that uses CSI ephemeral storage:

```
kind: Pod
apiVersion: v1
metadata:
  name: my-csi-app
spec:
  containers:
    - name: my-frontend
      image: busybox
      volumeMounts:
        - mountPath: "/data"
          name: my-csi-inline-vol
          command: [ "sleep", "1000000" ]
  volumes:
    - name: my-csi-inline-vol
      csi:
        driver: inline.storage.kubernetes.io
        volumeAttributes:
          foo: bar
```

The `volumeAttributes` determine what volume is prepared by the driver. These attributes are specific to each driver and not standardized. See the documentation of each CSI driver for further instructions.

As a cluster administrator, you can use a [PodSecurityPolicy](#) to control which CSI drivers can be used in a Pod, specified with the [allowedCSIDrivers field](#).

Generic ephemeral volumes

FEATURE STATE: Kubernetes v1.19 [alpha]

This feature requires the `GenericEphemeralVolume` [feature gate](#) to be enabled. Because this is an alpha feature, it is disabled by default.

Generic ephemeral volumes are similar to `emptyDir` volumes, just more flexible:

- Storage can be local or network-attached.
- Volumes can have a fixed size that Pods are not able to exceed.
- Volumes may have some initial data, depending on the driver and parameters.
- Typical operations on volumes are supported assuming that the driver supports them, including ([snapshotting](#), [cloning](#), [resizing](#), and [storage capacity tracking](#)).

Example:

```
kind: Pod
apiVersion: v1
metadata:
  name: my-app
spec:
  containers:
    - name: my-frontend
      image: busybox
      volumeMounts:
        - mountPath: "/scratch"
          name: scratch-volume
      command: [ "sleep", "1000000" ]
  volumes:
    - name: scratch-volume
      ephemeral:
        volumeClaimTemplate:
          metadata:
            labels:
              type: my-frontend-volume
      spec:
        accessModes: [ "ReadWriteOnce" ]
        storageClassName: "scratch-storage-class"
        resources:
          requests:
            storage: 1Gi
```

Lifecycle and PersistentVolumeClaim

The key design idea is that the [parameters for a volume claim](#) are allowed inside a volume source of the Pod. Labels, annotations and the whole set of fields for a PersistentVolumeClaim are supported. When such a Pod gets created, the ephemeral volume controller then creates an actual PersistentVolumeClaim object in the same namespace as the Pod and ensures that the PersistentVolumeClaim gets deleted when the Pod gets deleted.

That triggers volume binding and/or provisioning, either immediately if the [StorageClass](#) uses immediate volume binding or when the Pod is tentatively scheduled onto a node ([WaitForFirstConsumer](#) volume binding mode). The latter is recommended for generic ephemeral volumes because then the scheduler is free to choose a suitable node for the Pod. With immediate binding, the scheduler is forced to select a node that has access to the volume once it is available.

In terms of [resource ownership](#), a Pod that has generic ephemeral storage is the owner of the PersistentVolumeClaim(s) that provide that ephemeral storage. When the Pod is deleted, the Kubernetes garbage collector deletes the PVC, which then usually triggers deletion of the volume because the default reclaim policy of storage classes is to delete volumes. You can create quasi-ephemeral local storage using a StorageClass with a reclaim policy of

retain: the storage outlives the Pod, and in this case you need to ensure that volume clean up happens separately.

While these PVCs exist, they can be used like any other PVC. In particular, they can be referenced as data source in volume cloning or snapshotting. The PVC object also holds the current status of the volume.

PersistentVolumeClaim naming

Naming of the automatically created PVCs is deterministic: the name is a combination of Pod name and volume name, with a hyphen (-) in the middle. In the example above, the PVC name will be my-app-scratch-volume. This deterministic naming makes it easier to interact with the PVC because one does not have to search for it once the Pod name and volume name are known.

The deterministic naming also introduces a potential conflict between different Pods (a Pod "pod-a" with volume "scratch" and another Pod with name "pod" and volume "a-scratch" both end up with the same PVC name "pod-a-scratch") and between Pods and manually created PVCs.

Such conflicts are detected: a PVC is only used for an ephemeral volume if it was created for the Pod. This check is based on the ownership relationship. An existing PVC is not overwritten or modified. But this does not resolve the conflict because without the right PVC, the Pod cannot start.

Caution: Take care when naming Pods and volumes inside the same namespace, so that these conflicts can't occur.

Security

Enabling the GenericEphemeralVolume feature allows users to create PVCs indirectly if they can create Pods, even if they do not have permission to create PVCs directly. Cluster administrators must be aware of this. If this does not fit their security model, they have two choices:

- Explicitly disable the feature through the feature gate, to avoid being surprised when some future Kubernetes version enables it by default.
- Use a [Pod Security Policy](#) where the volumes list does not contain the ephemeral volume type.

The normal namespace quota for PVCs in a namespace still applies, so even if users are allowed to use this new mechanism, they cannot use it to circumvent other policies.

What's next

Ephemeral volumes managed by kubelet

See [local ephemeral storage](#).

CSI ephemeral volumes

- For more information on the design, see the [Ephemeral Inline CSI volumes KEP](#).
- For more information on further development of this feature, see the [enhancement tracking issue #596](#).

Generic ephemeral volumes

- For more information on the design, see the [Generic ephemeral inline volumes KEP](#).
- For more information on further development of this feature, see the [enhancement tracking issue #1698](#).

Feedback

Was this page helpful?

Yes No

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

Last modified November 28, 2020 at 12:43 PM PST: [Fix link to CSI ephemeral volumes \(01d4a0a4d\)](#)

[Edit this page](#) [Create child page](#) [Create an issue](#)

- [Types of ephemeral volumes](#)
- [CSI ephemeral volumes](#)
- [Generic ephemeral volumes](#)
- [Lifecycle and PersistentVolumeClaim](#)
- [PersistentVolumeClaim naming](#)
- [Security](#)
- [What's next](#)
 - [Ephemeral volumes managed by kubelet](#)
 - [CSI ephemeral volumes](#)
 - [Generic ephemeral volumes](#)

Node-specific Volume Limits

This page describes the maximum number of volumes that can be attached to a Node for various cloud providers.

Cloud providers like Google, Amazon, and Microsoft typically have a limit on how many volumes can be attached to a Node. It is important for Kubernetes to respect those limits. Otherwise, Pods scheduled on a Node could get stuck waiting for volumes to attach.

Kubernetes default limits

The Kubernetes scheduler has default limits on the number of volumes that can be attached to a Node:

Cloud service	Maximum volumes per Node
Amazon Elastic Block Store (EBS)	39
Google Persistent Disk	16
Microsoft Azure Disk Storage	16

Custom limits

You can change these limits by setting the value of the `KUBE_MAX_PD_VOLS` environment variable, and then starting the scheduler. CSI drivers might have a different procedure, see their documentation on how to customize their limits.

Use caution if you set a limit that is higher than the default limit. Consult the cloud provider's documentation to make sure that Nodes can actually support the limit you set.

The limit applies to the entire cluster, so it affects all Nodes.

Dynamic volume limits

FEATURE STATE: Kubernetes v1.17 [stable]

Dynamic volume limits are supported for following volume types.

- Amazon EBS
- Google Persistent Disk
- Azure Disk
- CSI

For volumes managed by in-tree volume plugins, Kubernetes automatically determines the Node type and enforces the appropriate maximum number of volumes for the node. For example:

- On [Google Compute Engine](#), up to 127 volumes can be attached to a node, [depending on the node type](#).
- For Amazon EBS disks on M5,C5,R5,T3 and Z1D instance types, Kubernetes allows only 25 volumes to be attached to a Node. For other instance types on [Amazon Elastic Compute Cloud \(EC2\)](#), Kubernetes allows 39 volumes to be attached to a Node.
- On Azure, up to 64 disks can be attached to a node, depending on the node type. For more details, refer to [Sizes for virtual machines in Azure](#).

If a CSI storage driver advertises a maximum number of volumes for a Node (using `NodeGetInfo`), the [kube-scheduler](#) honors that limit. Refer to the [CSI specifications](#) for details.

- For volumes managed by in-tree plugins that have been migrated to a CSI driver, the maximum number of volumes will be the one reported by the CSI driver.

Feedback

Was this page helpful?

Yes No

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

Last modified May 30, 2020 at 3:10 PM PST: [add en pages \(ecc27bbbe\)](#)
[Edit this page](#) [Create child page](#) [Create an issue](#)

- [Kubernetes default limits](#)
- [Custom limits](#)
- [Dynamic volume limits](#)

Configuration

Resources that Kubernetes provides for configuring Pods.

[**Configuration Best Practices**](#)

[**ConfigMaps**](#)

[**Secrets**](#)

[**Managing Resources for Containers**](#)

[**Organizing Cluster Access Using kubeconfig Files**](#)

[**Pod Priority and Preemption**](#)

Configuration Best Practices

This document highlights and consolidates configuration best practices that are introduced throughout the user guide, Getting Started documentation, and examples.

This is a living document. If you think of something that is not on this list but might be useful to others, please don't hesitate to file an issue or submit a PR.

General Configuration Tips

- When defining configurations, specify the latest stable API version.
- Configuration files should be stored in version control before being pushed to the cluster. This allows you to quickly roll back a configuration change if necessary. It also aids cluster re-creation and restoration.
- Write your configuration files using YAML rather than JSON. Though these formats can be used interchangeably in almost all scenarios, YAML tends to be more user-friendly.
- Group related objects into a single file whenever it makes sense. One file is often easier to manage than several. See the [`guestbook-all-in-one.yaml`](#) file as an example of this syntax.
- Note also that many `kubectl` commands can be called on a directory. For example, you can call `kubectl apply` on a directory of config files.
- Don't specify default values unnecessarily: simple, minimal configuration will make errors less likely.
- Put object descriptions in annotations, to allow better introspection.

"Naked" Pods versus ReplicaSets, Deployments, and Jobs

- Don't use naked Pods (that is, Pods not bound to a [ReplicaSet](#) or [Deployment](#)) if you can avoid it. Naked Pods will not be rescheduled in the event of a node failure.

A Deployment, which both creates a ReplicaSet to ensure that the desired number of Pods is always available, and specifies a strategy to replace Pods (such as [RollingUpdate](#)), is almost always preferable to creating Pods directly, except for some explicit [restartPolicy: Never](#) scenarios. A [Job](#) may also be appropriate.

Services

- Create a [Service](#) before its corresponding backend workloads (Deployments or ReplicaSets), and before any workloads that need to access it. When Kubernetes starts a container, it provides environment variables pointing to all the Services which were running when the container was started. For example, if a Service named `foo` exists, all containers will get the following variables in their initial environment:

```
FOO_SERVICE_HOST=<the host the Service is running on>  
FOO_SERVICE_PORT=<the port the Service is running on>
```

This does imply an ordering requirement - any Service that a Pod wants to access must be created before the Pod itself, or else the environment variables will not be populated. DNS does not have this restriction.

- An optional (though strongly recommended) [cluster add-on](#) is a DNS server. The DNS server watches the Kubernetes API for new Services and creates a set of DNS records for each. If DNS has been enabled throughout the cluster then all Pods should be able to do name resolution of Services automatically.
- Don't specify a `hostPort` for a Pod unless it is absolutely necessary. When you bind a Pod to a `hostPort`, it limits the number of places the Pod can be scheduled, because each `<hostIP, hostPort, protocol>` combination must be unique. If you don't specify the `hostIP` and `proto col` explicitly, Kubernetes will use `0.0.0.0` as the default `hostIP` and `TCP` as the default `protocol`.

If you only need access to the port for debugging purposes, you can use the [apiserver proxy](#) or [kubectl port-forward](#).

If you explicitly need to expose a Pod's port on the node, consider using a [NodePort](#) Service before resorting to `hostPort`.

- Avoid using `hostNetwork`, for the same reasons as `hostPort`.
- Use [headless Services](#) (which have a `ClusterIP` of `None`) for easy service discovery when you don't need `kube-proxy` load balancing.

Using Labels

- Define and use [labels](#) that identify **semantic attributes** of your application or Deployment, such as `{ app: myapp, tier: frontend, phase: test, deployment: v3 }`. You can use these labels to select the appropriate Pods for other resources; for example, a Service that selects all `tier: frontend` Pods, or all `phase: test` components of `app : myapp`. See the [guestbook](#) app for examples of this approach.

A Service can be made to span multiple Deployments by omitting release-specific labels from its selector. [Deployments](#) make it easy to update a running service without downtime.

A desired state of an object is described by a Deployment, and if changes to that spec are applied, the deployment controller changes the actual state to the desired state at a controlled rate.

- You can manipulate labels for debugging. Because Kubernetes controllers (such as ReplicaSet) and Services match to Pods using selector labels, removing the relevant labels from a Pod will stop it from

being considered by a controller or from being served traffic by a Service. If you remove the labels of an existing Pod, its controller will create a new Pod to take its place. This is a useful way to debug a previously "live" Pod in a "quarantine" environment. To interactively remove or add labels, use [kubectl label](#).

Container Images

The [imagePullPolicy](#) and the tag of the image affect when the [kubelet](#) attempts to pull the specified image.

- *imagePullPolicy: IfNotPresent: the image is pulled only if it is not already present locally.*
- *imagePullPolicy: Always: every time the kubelet launches a container, the kubelet queries the container image registry to resolve the name to an image digest. If the kubelet has a container image with that exact digest cached locally, the kubelet uses its cached image; otherwise, the kubelet downloads (pulls) the image with the resolved digest, and uses that image to launch the container.*
- *imagePullPolicy is omitted and either the image tag is :latest or it is omitted: Always is applied.*
- *imagePullPolicy is omitted and the image tag is present but not :latest: IfNotPresent is applied.*
- *imagePullPolicy: Never: the image is assumed to exist locally. No attempt is made to pull the image.*

Note: To make sure the container always uses the same version of the image, you can specify its [digest](#); replace <image-name>:<tag> with <image-name>@<digest> (for example, `image@sha256:45b23d ee08af5e43a7fea6c4cf9c25ccf269ee113168c19722f87876677c5c b2`). The digest uniquely identifies a specific version of the image, so it is never updated by Kubernetes unless you change the digest value.

Note: You should avoid using the :latest tag when deploying containers in production as it is harder to track which version of the image is running and more difficult to roll back properly.

Note: The caching semantics of the underlying image provider make even `imagePullPolicy: Always` efficient. With Docker, for example, if the image already exists, the pull attempt is fast because all image layers are cached and no image download is needed.

Using kubectl

- Use `kubectl apply -f <directory>`. This looks for Kubernetes configuration in all `.yaml`, `.yml`, and `.json` files in `<directory>` and passes it to `apply`.
- Use label selectors for `get` and `delete` operations instead of specific object names. See the sections on [label selectors](#) and [using labels effectively](#).
- Use `kubectl create deployment` and `kubectl expose` to quickly create single-container Deployments and Services. See [Use a Service to Access an Application in a Cluster](#) for an example.

Feedback

Was this page helpful?

Yes No

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

Last modified July 17, 2020 at 3:46 PM PST: [Replace reference to redirect entries \(1\) \(0bdcd44e6\)](#)
[Edit this page](#) [Create child page](#) [Create an issue](#)

- [General Configuration Tips](#)
- ["Naked" Pods versus ReplicaSets, Deployments, and Jobs](#)
- [Services](#)
- [Using Labels](#)
- [Container Images](#)
- [Using kubectl](#)

ConfigMaps

A ConfigMap is an API object used to store non-confidential data in key-value pairs. [Pods](#) can consume ConfigMaps as environment variables, command-line arguments, or as configuration files in a [volume](#).

A ConfigMap allows you to decouple environment-specific configuration from your [container images](#), so that your applications are easily portable.

Caution: ConfigMap does not provide secrecy or encryption. If the data you want to store are confidential, use a [Secret](#) rather than a ConfigMap, or use additional (third party) tools to keep your data private.

Motivation

Use a *ConfigMap* for setting configuration data separately from application code.

For example, imagine that you are developing an application that you can run on your own computer (for development) and in the cloud (to handle real traffic). You write the code to look in an environment variable named `DATABASE_HOST`. Locally, you set that variable to `localhost`. In the cloud, you set it to refer to a Kubernetes [Service](#) that exposes the database component to your cluster. This lets you fetch a container image running in the cloud and debug the exact same code locally if needed.

A *ConfigMap* is not designed to hold large chunks of data. The data stored in a *ConfigMap* cannot exceed 1 MiB. If you need to store settings that are larger than this limit, you may want to consider mounting a volume or use a separate database or file service.

ConfigMap object

A *ConfigMap* is an API [object](#) that lets you store configuration for other objects to use. Unlike most Kubernetes objects that have a `spec`, a *ConfigMap* has `data` and `binaryData` fields. These fields accept key-value pairs as their values. Both the `data` field and the `binaryData` are optional. The `data` field is designed to contain UTF-8 byte sequences while the `binaryData` field is designed to contain binary data.

The name of a *ConfigMap* must be a valid [DNS subdomain name](#).

Each key under the `data` or the `binaryData` field must consist of alphanumeric characters, `-`, `_` or `.`. The keys stored in `data` must not overlap with the keys in the `binaryData` field.

Starting from v1.19, you can add an `immutable` field to a *ConfigMap* definition to create an [immutable ConfigMap](#).

ConfigMaps and Pods

You can write a `Pod spec` that refers to a *ConfigMap* and configures the container(s) in that Pod based on the data in the *ConfigMap*. The Pod and the *ConfigMap* must be in the same [namespace](#).

Here's an example *ConfigMap* that has some keys with single values, and other keys where the value looks like a fragment of a configuration format.

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: game-demo
data:
  # property-like keys; each key maps to a simple value
```

```

player_initial_lives: "3"
ui_properties_file_name: "user-interface.properties"

# file-like keys
game.properties: |
  enemy.types=aliens,monsters
  player.maximum-lives=5
user-interface.properties: |
  color.good=purple
  color.bad=yellow
  allow.textmode=true

```

There are four different ways that you can use a ConfigMap to configure a container inside a Pod:

1. Inside a container command and args
2. Environment variables for a container
3. Add a file in read-only volume, for the application to read
4. Write code to run inside the Pod that uses the Kubernetes API to read a ConfigMap

These different methods lend themselves to different ways of modeling the data being consumed. For the first three methods, the [kubelet](#) uses the data from the ConfigMap when it launches container(s) for a Pod.

The fourth method means you have to write code to read the ConfigMap and its data. However, because you're using the Kubernetes API directly, your application can subscribe to get updates whenever the ConfigMap changes, and react when that happens. By accessing the Kubernetes API directly, this technique also lets you access a ConfigMap in a different namespace.

Here's an example Pod that uses values from game-demo to configure a Pod:

```

apiVersion: v1
kind: Pod
metadata:
  name: configmap-demo-pod
spec:
  containers:
    - name: demo
      image: alpine
      command: ["sleep", "3600"]
      env:
        # Define the environment variable
        - name: PLAYER_INITIAL_LIVES # Notice that the case is
different here
                                                # from the key name in the
ConfigMap.
        valueFrom:
          configMapKeyRef:
            name: game-demo
            # The ConfigMap this
value comes from.
            key: player_initial_lives # The key to fetch.

```

```

- name: UI_PROPERTIES_FILE_NAME
valueFrom:
  configMapKeyRef:
    name: game-demo
    key: ui_properties_file_name
volumeMounts:
- name: config
  mountPath: "/config"
  readOnly: true
volumes:
  # You set volumes at the Pod level, then mount them into
  containers inside that Pod
- name: config
  configMap:
    # Provide the name of the ConfigMap you want to mount.
    name: game-demo
    # An array of keys from the ConfigMap to create as files
    items:
- key: "game.properties"
  path: "game.properties"
- key: "user-interface.properties"
  path: "user-interface.properties"

```

A ConfigMap doesn't differentiate between single line property values and multi-line file-like values. What matters is how Pods and other objects consume those values.

For this example, defining a volume and mounting it inside the demo container as /config creates two files, /config/game.properties and /config/user-interface.properties, even though there are four keys in the ConfigMap. This is because the Pod definition specifies an items array in the volumes section. If you omit the items array entirely, every key in the ConfigMap becomes a file with the same name as the key, and you get 4 files.

Using ConfigMaps

ConfigMaps can be mounted as data volumes. ConfigMaps can also be used by other parts of the system, without being directly exposed to the Pod. For example, ConfigMaps can hold data that other parts of the system should use for configuration.

The most common way to use ConfigMaps is to configure settings for containers running in a Pod in the same namespace. You can also use a ConfigMap separately.

For example, you might encounter [addons](#) or [operators](#) that adjust their behavior based on a ConfigMap.

Using ConfigMaps as files from a Pod

To consume a ConfigMap in a volume in a Pod:

1. Create a ConfigMap or use an existing one. Multiple Pods can reference the same ConfigMap.
2. Modify your Pod definition to add a volume under `.spec.volumes[]`. Name the volume anything, and have a `.spec.volumes[].configMap.name` field set to reference your ConfigMap object.
3. Add a `.spec.containers[].volumeMounts[]` to each container that needs the ConfigMap. Specify `.spec.containers[].volumeMounts[].readOnly = true` and `.spec.containers[].volumeMounts[].mountPath` to an unused directory name where you would like the ConfigMap to appear.
4. Modify your image or command line so that the program looks for files in that directory. Each key in the ConfigMap data map becomes the filename under `mountPath`.

This is an example of a Pod that mounts a ConfigMap in a volume:

```
apiVersion: v1
kind: Pod
metadata:
  name: mypod
spec:
  containers:
    - name: mypod
      image: redis
      volumeMounts:
        - name: foo
          mountPath: "/etc/foo"
          readOnly: true
  volumes:
    - name: foo
      configMap:
        name: myconfigmap
```

Each ConfigMap you want to use needs to be referred to in `.spec.volumes`.

If there are multiple containers in the Pod, then each container needs its own `volumeMounts` block, but only one `.spec.volumes` is needed per ConfigMap.

Mounted ConfigMaps are updated automatically

When a ConfigMap currently consumed in a volume is updated, projected keys are eventually updated as well. The kubelet checks whether the mounted ConfigMap is fresh on every periodic sync. However, the kubelet uses its local cache for getting the current value of the ConfigMap. The type of the cache is configurable using the `ConfigMapAndSecretChangeDetectionStrategy` field in the [KubeletConfiguration struct](#). A ConfigMap can be either propagated by watch (default), ttl-based, or simply redirecting all

requests directly to the API server. 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 the kubelet sync period + cache propagation delay, where the cache propagation delay depends on the chosen cache type (it equals to watch propagation delay, ttl of cache, or zero correspondingly).

ConfigMaps consumed as environment variables are not updated automatically and require a pod restart.

Immutable ConfigMaps

FEATURE STATE: Kubernetes v1.19 [beta]

The Kubernetes beta feature Immutable Secrets and ConfigMaps provides an option to set individual Secrets and ConfigMaps as immutable. For clusters that extensively use ConfigMaps (at least tens of thousands of unique ConfigMap to Pod mounts), preventing changes to their data has the following advantages:

- *protects you from accidental (or unwanted) updates that could cause applications outages*
- *improves performance of your cluster by significantly reducing load on kube-apiserver, by closing watches for ConfigMaps marked as immutable.*

This feature is controlled by the `ImmutableEphemeralVolumes` [feature gate](#). You can create an immutable ConfigMap by setting the `immutable` field to `true`. For example:

```
apiVersion: v1
kind: ConfigMap
metadata:
  ...
data:
  ...
immutable: true
```

Once a ConfigMap is marked as immutable, it is not possible to revert this change nor to mutate the contents of the `data` or the `binaryData` field. You can only delete and recreate the ConfigMap. Because existing Pods maintain a mount point to the deleted ConfigMap, it is recommended to recreate these pods.

What's next

- Read about [Secrets](#).
- Read [Configure a Pod to Use a ConfigMap](#).
- Read [The Twelve-Factor App](#) to understand the motivation for separating code from configuration.

Feedback

Was this page helpful?

Yes No

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

Last modified November 18, 2020 at 10:40 AM PST: [Update configmap.md \(97629eba2\)](#)

[Edit this page](#) [Create child page](#) [Create an issue](#)

- [Motivation](#)
- [ConfigMap object](#)
- [ConfigMaps and Pods](#)
- [Using ConfigMaps](#)
 - [Using ConfigMaps as files from a Pod](#)
- [Immutable ConfigMaps](#)
- [What's next](#)

Secrets

Kubernetes Secrets let you store and manage sensitive information, such as passwords, OAuth tokens, and ssh keys. Storing confidential information in a Secret is safer and more flexible than putting it verbatim in a [Pod](#) definition or in a [container image](#). See [Secrets design document](#) for more information.

A Secret is an object that contains a small amount of sensitive data such as a password, a token, or a key. Such information might otherwise be put in a Pod specification or in an image. Users can create Secrets and the system also creates some Secrets.

Overview of Secrets

To use a Secret, a Pod needs to reference the Secret. A Secret can be used with a Pod in three ways:

- As [files](#) in a [volume](#) mounted on one or more of its containers.
- As [container environment variable](#).
- By the [kubelet when pulling images](#) for the Pod.

The name of a Secret object must be a valid [DNS subdomain name](#). You can specify the `data` and/or the `stringData` field when creating a configuration file for a Secret. The `data` and the `stringData` fields are optional. The values for all keys in the `data` field have to be base64-encoded strings. If the conversion to base64 string is not desirable, you can choose to specify the `stringData` field instead, which accepts arbitrary strings as values.

The keys of `data` and `stringData` must consist of alphanumeric characters, `-`, `_` or `.`. All key-value pairs in the `stringData` field are internally merged into the `data` field. If a key appears in both the `data` and the `stringData` field, the value specified in the `stringData` field takes precedence.

Types of Secret

When creating a Secret, you can specify its type using the `type` field of the [Secret](#) resource, or certain equivalent `kubectl` command line flags (if available). The Secret type is used to facilitate programmatic handling of the Secret data.

Kubernetes provides several builtin types for some common usage scenarios. These types vary in terms of the validations performed and the constraints Kubernetes imposes on them.

Builtin Type	Usage
Opaque	arbitrary user-defined data
<code>kubernetes.io/service-account-token</code>	service account token
<code>kubernetes.io/dockercfg</code>	serialized <code>~/.dockercfg</code> file
<code>kubernetes.io/dockerconfigjson</code>	serialized <code>~/.docker/config.json</code> file
<code>kubernetes.io/basic-auth</code>	credentials for basic authentication
<code>kubernetes.io/ssh-auth</code>	credentials for SSH authentication
<code>kubernetes.io/tls</code>	data for a TLS client or server
<code>bootstrap.kubernetes.io/token</code>	bootstrap token data

You can define and use your own Secret type by assigning a non-empty string as the `type` value for a Secret object. An empty string is treated as an `Opaque` type. Kubernetes doesn't impose any constraints on the type name. However, if you are using one of the builtin types, you must meet all the requirements defined for that type.

Opaque secrets

`Opaque` is the default Secret type if omitted from a Secret configuration file. When you create a Secret using `kubectl`, you will use the `generic` subcommand to indicate an `Opaque` Secret type. For example, the following command creates an empty Secret of type `Opaque`.

```
kubectl create secret generic empty-secret  
kubectl get secret empty-secret
```

The output looks like:

NAME	TYPE	DATA	AGE
empty-secret	Opaque	0	2m6s

The *DATA* column shows the number of data items stored in the Secret. In this case, 0 means we have just created an empty Secret.

Service account token Secrets

A `kubernetes.io/service-account-token` type of Secret is used to store a token that identifies a service account. When using this Secret type, you need to ensure that the `kubernetes.io/service-account.name` annotation is set to an existing service account name. A Kubernetes controller fills in some other fields such as the `kubernetes.io/service-account.uid` annotation and the `token` key in the `data` field set to actual token content.

The following example configuration declares a service account token Secret:

```
apiVersion: v1
kind: Secret
metadata:
  name: secret-sa-sample
  annotations:
    kubernetes.io/service-account.name: "sa-name"
type: kubernetes.io/service-account-token
data:
  # You can include additional key value pairs as you do with
  Opaque Secrets
  extra: YmFyCg==
```

When creating a Pod, Kubernetes automatically creates a service account Secret and automatically modifies your Pod to use this Secret. The service account token Secret contains credentials for accessing the API.

The automatic creation and use of API credentials can be disabled or overridden if desired. However, if all you need to do is securely access the API server, this is the recommended workflow.

See the [ServiceAccount](#) documentation for more information on how service accounts work. You can also check the `automountServiceAccountToken` field and the `serviceAccountName` field of the [Pod](#) for information on referencing service account from Pods.

Docker config Secrets

You can use one of the following type values to create a Secret to store the credentials for accessing a Docker registry for images.

- `kubernetes.io/dockercfg`
- `kubernetes.io/dockerconfigjson`

The `kubernetes.io/dockercfg` type is reserved to store a serialized `~/.dockercfg` which is the legacy format for configuring Docker command line. When using this Secret type, you have to ensure the Secret data field

contains a `.dockercfg` key whose value is content of a `~/.dockercfg` file encoded in the base64 format.

The `kubernetes.io/dockerconfigjson` type is designed for storing a serialized JSON that follows the same format rules as the `~/.docker/config.json` file which is a new format for `~/.dockercfg`. When using this Secret type, the `data` field of the Secret object must contain a `.dockerconfigjson` key, in which the content for the `~/.docker/config.json` file is provided as a base64 encoded string.

Below is an example for a `kubernetes.io/dockercfg` type of Secret:

```
apiVersion: v1
kind: Secret
metadata:
  name: secret-dockercfg
type: kubernetes.io/dockercfg
data:
  .dockercfg: |
    "<base64 encoded ~/.dockercfg file>"
```

Note: If you do not want to perform the base64 encoding, you can choose to use the `stringData` field instead.

When you create these types of Secrets using a manifest, the API server checks whether the expected key does exists in the `data` field, and it verifies if the value provided can be parsed as a valid JSON. The API server doesn't validate if the JSON actually is a Docker config file.

When you do not have a Docker config file, or you want to use `kubectl` to create a Docker registry Secret, you can do:

```
kubectl create secret docker-registry secret-tiger-docker \
--docker-username=tiger \
--docker-password=pass113 \
--docker-email=tiger@acme.com
```

This command creates a Secret of type `kubernetes.io/dockerconfigjson`. If you dump the `.dockerconfigjson` content from the `data` field, you will get the following JSON content which is a valid Docker configuration created on the fly:

```
{
  "auths": {
    "https://index.docker.io/v1/": {
      "username": "tiger",
      "password": "pass113",
      "email": "tiger@acme.com",
      "auth": "dGlnZXI6cGFzcExMw=="
    }
  }
}
```

Basic authentication Secret

The `kubernetes.io/basic-auth` type is provided for storing credentials needed for basic authentication. When using this Secret type, the `data` field of the Secret must contain the following two keys:

- `username`: the user name for authentication;
- `password`: the password or token for authentication.

Both values for the above two keys are base64 encoded strings. You can, of course, provide the clear text content using the `stringData` for Secret creation.

The following YAML is an example config for a basic authentication Secret:

```
apiVersion: v1
kind: Secret
metadata:
  name: secret-basic-auth
type: kubernetes.io/basic-auth
stringData:
  username: admin
  password: t0p-Secret
```

The basic authentication Secret type is provided only for user's convenience. You can create an Opaque for credentials used for basic authentication. However, using the builtin Secret type helps unify the formats of your credentials and the API server does verify if the required keys are provided in a Secret configuration.

SSH authentication secrets

The builtin type `kubernetes.io/ssh-auth` is provided for storing data used in SSH authentication. When using this Secret type, you will have to specify a `ssh-privatekey` key-value pair in the `data` (or `stringData`) field as the SSH credential to use.

The following YAML is an example config for a SSH authentication Secret:

```
apiVersion: v1
kind: Secret
metadata:
  name: secret-ssh-auth
type: kubernetes.io/ssh-auth
data:
  # the data is abbreviated in this example
  ssh-privatekey: |
    MIIEpQIBAAKCAQEAlqb/Y ...
```

The SSH authentication Secret type is provided only for user's convenience. You can create an Opaque for credentials used for SSH authentication. However, using the builtin Secret type helps unify the formats of your

credentials and the API server does verify if the required keys are provided in a Secret configuration.

Caution: SSH private keys do not establish trusted communication between an SSH client and host server on their own. A secondary means of establishing trust is needed to mitigate "man in the middle" attacks, such as a `known_hosts` file added to a `ConfigMap`.

TLS secrets

Kubernetes provides a builtin Secret type `kubernetes.io/tls` for storing a certificate and its associated key that are typically used for TLS. This data is primarily used with TLS termination of the Ingress resource, but may be used with other resources or directly by a workload. When using this type of Secret, the `tls.key` and the `tls.crt` key must be provided in the `data` (or `stringData`) field of the Secret configuration, although the API server doesn't actually validate the values for each key.

The following YAML contains an example config for a TLS Secret:

```
apiVersion: v1
kind: Secret
metadata:
  name: secret-tls
type: kubernetes.io/tls
data:
  # the data is abbreviated in this example
  tls.crt: |
    MIIC2DCCAcCgAwIBAgIBATANBgkqh ...
  tls.key: |
    MIIEpgIBAAKCAQEA7yn3bRHQ5FHMQ ...
```

The TLS Secret type is provided for user's convenience. You can create an opaque for credentials used for TLS server and/or client. However, using the builtin Secret type helps ensure the consistency of Secret format in your project; the API server does verify if the required keys are provided in a Secret configuration.

When creating a TLS Secret using `kubectl`, you can use the `tls` subcommand as shown in the following example:

```
kubectl create secret tls my-tls-secret \
--cert=path/to/cert/file \
--key=path/to/key/file
```

The public/private key pair must exist before hand. The public key certificate for `--cert` must be .PEM encoded (Base64-encoded DER format), and match the given private key for `--key`. The private key must be in what is commonly called PEM private key format, unencrypted. In both cases, the initial and the last lines from PEM (for example, `-----BEGIN CERTIFICATE-----` and `-----END CERTIFICATE-----` for a certificate) are not included.

Bootstrap token Secrets

A bootstrap token Secret can be created by explicitly specifying the Secret type to `bootstrap.kubernetes.io/token`. This type of Secret is designed for tokens used during the node bootstrap process. It stores tokens used to sign well known ConfigMaps.

A bootstrap token Secret is usually created in the `kube-system` namespace and named in the form `bootstrap-token-<token-id>` where `<token-id>` is a 6 character string of the token ID.

As a Kubernetes manifest, a bootstrap token Secret might look like the following:

```
apiVersion: v1
kind: Secret
metadata:
  name: bootstrap-token-5emitj
  namespace: kube-system
type: bootstrap.kubernetes.io/token
data:
  auth-extra-groups: c3lzdGVtOmJvb3RzdHJhcHBlcnM6a3ViZwFkbTpkZWZh
dWx0LW5vZGUtG9rZW4=
  expiration: MjAyMC0wOS0xM1QwNDozOToxMFo=
  token-id: NwVtaXRq
  token-secret: a3E0Z2lodnN6emduMXAwcg==
  usage-bootstrap-authentication: dHJ1ZQ==
  usage-bootstrap-signing: dHJ1ZQ==
```

A bootstrap type Secret has the following keys specified under `data`:

- `token_id`: A random 6 character string as the token identifier. Required.
- `token-secret`: A random 16 character string as the actual token secret. Required.
- `description`: A human-readable string that describes what the token is used for. Optional.
- `expiration`: An absolute UTC time using RFC3339 specifying when the token should be expired. Optional.
- `usage-bootstrap-<usage>`: A boolean flag indicating additional usage for the bootstrap token.
- `auth-extra-groups`: A comma-separated list of group names that will be authenticated as in addition to the `system:bootstrappers` group.

The above YAML may look confusing because the values are all in base64 encoded strings. In fact, you can create an identical Secret using the following YAML:

```
apiVersion: v1
kind: Secret
metadata:
  # Note how the Secret is named
  name: bootstrap-token-5emitj
```

```
# A bootstrap token Secret usually resides in the kube-system
namespace
  namespace: kube-system
type: bootstrap.kubernetes.io/token
stringData:
  auth-extra-groups: "system:bootstrappers:kubeadm:default-node-
token"
    expiration: "2020-09-13T04:39:10Z"
    # This token ID is used in the name
    token-id: "5emitj"
    token-secret: "kq4gihvsszgn1p0r"
    # This token can be used for authentication
    usage-bootstrap-authentication: "true"
    # and it can be used for signing
    usage-bootstrap-signing: "true"
```

Creating a Secret

There are several options to create a Secret:

- [create Secret using kubectl command](#)
- [create Secret from config file](#)
- [create Secret using kustomize](#)

Editing a Secret

An existing Secret may be edited with the following command:

```
kubectl edit secrets mysecret
```

This will open the default configured editor and allow for updating the base64 encoded Secret values in the data field:

```
# 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 this file will be
# reopened with the relevant failures.
#
apiVersion: v1
data:
  username: YwRtaW4=
  password: MwYyZDFlMmU2N2Rm
kind: Secret
metadata:
  annotations:
    kubectl.kubernetes.io/last-applied-configuration: { ... }
  creationTimestamp: 2016-01-22T18:41:56Z
  name: mysecret
  namespace: default
  resourceVersion: "164619"
```

```
uid: cfee02d6-c137-11e5-8d73-42010af00002  
type: Opaque
```

Using Secrets

Secrets can be mounted as data volumes or exposed as [environment variables](#) to be used by a container in a Pod. Secrets can also be used by other parts of the system, without being directly exposed to the Pod. For example, Secrets can hold credentials that other parts of the system should use to interact with external systems on your behalf.

Using Secrets as files from a Pod

To consume a Secret in a volume in a Pod:

1. Create a secret or use an existing one. Multiple Pods can reference the same secret.
2. Modify your Pod definition to add a volume under `.spec.volumes[]`. Name the volume anything, and have a `.spec.volumes[].secret.secretName` field equal to the name of the Secret object.
3. Add a `.spec.containers[].volumeMounts[]` to each container that needs the secret. Specify `.spec.containers[].volumeMounts[].readOnly = true` and `.spec.containers[].volumeMounts[].mountPath` to an unused directory name where you would like the secrets to appear.
4. Modify your image or command line so that the program looks for files in that directory. Each key in the secret data map becomes the filename under `mountPath`.

This is an example of a Pod that mounts a Secret in a volume:

```
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
```

Each Secret you want to use needs to be referred to in `.spec.volumes`.

If there are multiple containers in the Pod, then each container needs its own `volumeMounts` block, but only one `.spec.volumes` is needed per Secret.

You can package many files into one secret, or use many secrets, whichever is convenient.

Projection of Secret keys to specific paths

You can also control the paths within the volume where Secret keys are projected. You can 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
```

What will happen:

- *username* secret is stored under `/etc/foo/my-group/my-username` file instead of `/etc/foo/username`.
- *password* secret is not projected.

If `.spec.volumes[].secret.items` is used, 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.

Secret files permissions

You can set the 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 mode for the entire Secret volume and 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
```

Then, the secret will be mounted on `/etc/foo` and all the files created by the secret volume mount will have permission `0400`.

Note that the JSON spec doesn't support octal notation, so use the value `256` for `0400` permissions. If you use YAML instead of JSON for the Pod, you can use octal notation to specify permissions in a more natural way.

Note if you `kubectl exec` into the Pod, you need to follow the symlink to find the expected file mode. For example,

Check the secrets file mode on the pod.

```
kubectl exec mypod -it sh  
  
cd /etc/foo  
ls -l
```

The output is similar to this:

```
total 0  
lrwxrwxrwx 1 root root 15 May 18 00:18 password -> ..data/  
password  
lrwxrwxrwx 1 root root 15 May 18 00:18 username -> ..data/  
username
```

Follow the symlink to find the correct file mode.

```
cd /etc/foo/..data  
ls -l
```

The output is similar to this:

```
total 8  
-r----- 1 root root 12 May 18 00:18 password  
-r----- 1 root root 5 May 18 00:18 username
```

You can also use mapping, as in the previous example, and specify different permissions for different files 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
        items:
          - key: username
            path: my-group/my-username
            mode: 0777
```

In this case, the file resulting in `/etc/foo/my-group/my-username` will have permission value of `0777`. If you use JSON, owing to JSON limitations, you must specify the mode in decimal notation, `511`.

Note that this permission value might be displayed in decimal notation if you read it later.

Consuming Secret values from volumes

Inside the container that mounts a secret volume, the secret keys appear as files and the secret values are base64 decoded and stored inside these files. This is the result of commands executed inside the container from the example above:

```
ls /etc/foo/
```

The output is similar to:

```
username
password
```

```
cat /etc/foo/username
```

The output is similar to:

```
admin
```

```
cat /etc/foo/password
```

The output is similar to:

```
1f2d1e2e67df
```

The program in a container is responsible for reading the secrets from the files.

Mounted Secrets are updated automatically

When a secret currently consumed in a volume is updated, projected keys are eventually updated as well. The kubelet checks whether the mounted secret is fresh on every periodic sync. However, the kubelet uses its local cache for getting the current value of the Secret. The type of the cache is configurable using the `ConfigMapAndSecretChangeDetectionStrategy` field in the [KubeletConfiguration struct](#). A Secret can be either propagated by watch (default), ttl-based, or simply redirecting all requests directly to the API server. As a result, the total delay from the moment when the Secret is updated to the moment when new keys are projected to the Pod can be as long as the kubelet sync period + cache propagation delay, where the cache propagation delay depends on the chosen cache type (it equals to watch propagation delay, ttl of cache, or zero correspondingly).

Note: A container using a Secret as a [subPath](#) volume mount will not receive Secret updates.

Using Secrets as environment variables

To use a secret in an [environment variable](#) in a Pod:

1. Create a secret or use an existing one. Multiple Pods can reference the same secret.
2. Modify your Pod definition in each container that you wish to consume the value of a secret key to add an environment variable for each secret key you wish to consume. The environment variable that consumes the secret key should populate the secret's name and key in `env[].valueFrom.secretKeyRef`.
3. Modify your image and/or command line so that the program looks for values in the specified environment variables.

This is an example of a Pod that uses secrets from environment variables:

```
apiVersion: v1
kind: Pod
metadata:
  name: secret-env-pod
spec:
  containers:
    - name: mycontainer
      image: redis
      env:
        - name: SECRET_USERNAME
          valueFrom:
            secretKeyRef:
              name: mysecret
              key: username
        - name: SECRET_PASSWORD
          valueFrom:
            secretKeyRef:
              name: mysecret
```

```
key: password  
restartPolicy: Never
```

Consuming Secret Values from environment variables

Inside a container that consumes a secret in an environment variables, the secret keys appear as normal environment variables containing the base64 decoded values of the secret data. This is the result of commands executed inside the container from the example above:

```
echo $SECRET_USERNAME
```

The output is similar to:

```
admin
```

```
echo $SECRET_PASSWORD
```

The output is similar to:

```
1f2d1e2e67df
```

Environment variables are not updated after a secret update

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.

Immutable Secrets

FEATURE STATE: Kubernetes v1.19 [beta]

The Kubernetes beta feature *Immutable Secrets* and *ConfigMaps* provides an option to set individual *Secrets* and *ConfigMaps* as immutable. For clusters that extensively use *Secrets* (at least tens of thousands of unique *Secret* to *Pod* mounts), preventing changes to their data has the following advantages:

- protects you from accidental (or unwanted) updates that could cause applications outages
- improves performance of your cluster by significantly reducing load on *kube-apiserver*, by closing watches for secrets marked as immutable.

This feature is controlled by the *ImmutableEphemeralVolumes* [feature gate](#), which is enabled by default since v1.19. You can create an immutable *Secret* by setting the *immutable* field to *true*. For example,

```
apiVersion: v1  
kind: Secret  
metadata:  
  ...  
data:
```

```
    ...
  immutable: true
```

Note: Once a Secret or ConfigMap is marked as immutable, it is not possible to revert this change nor to mutate the contents of the data field. You can only delete and recreate the Secret. Existing Pods maintain a mount point to the deleted Secret - it is recommended to recreate these pods.

Using `imagePullSecrets`

The `imagePullSecrets` field is a list of references to secrets in the same namespace. You can use an `imagePullSecrets` to pass a secret that contains a Docker (or other) image registry password to the kubelet. The kubelet uses this information to pull a private image on behalf of your Pod. See the [PodSpec API](#) for more information about the `imagePullSecrets` field.

Manually specifying an `imagePullSecret`

You can learn how to specify `ImagePullSecrets` from the [container images documentation](#).

Arranging for `imagePullSecrets` to be automatically attached

You can manually create `imagePullSecrets`, and reference it from a ServiceAccount. Any Pods created with that ServiceAccount or created with that ServiceAccount by default, will get their `imagePullSecrets` field set to that of the service account. See [Add ImagePullSecrets to a service account](#) for a detailed explanation of that process.

Automatic mounting of manually created Secrets

Manually created secrets (for example, one containing a token for accessing a GitHub account) can be automatically attached to pods based on their service account. See [Injecting Information into Pods Using a PodPreset](#) for a detailed explanation of that process.

Details

Restrictions

Secret volume sources are validated to ensure that the specified object reference actually points to an object of type Secret. Therefore, a secret needs to be created before any Pods that depend on it.

Secret resources reside in a [namespace](#). Secrets can only be referenced by Pods in that same namespace.

Individual secrets are limited to 1MiB in size. This is to discourage creation of very large secrets which would exhaust the API server and kubelet memory. However, creation of many smaller secrets could also exhaust memory. More comprehensive limits on memory usage due to secrets is a planned feature.

The kubelet only supports the use of secrets for Pods where the secrets are obtained from the API server. This includes any Pods created using kubectl, or indirectly via a replication controller. It does not include Pods created as a result of the kubelet --manifest-url flag, its --config flag, or its REST API (these are not common ways to create Pods.)

Secrets must be created before they are consumed in Pods as environment variables unless they are marked as optional. References to secrets that do not exist will prevent the Pod from starting.

References (secretKeyRef field) to keys that do not exist in a named Secret will prevent the Pod from starting.

Secrets used to populate environment variables by the envFrom field that have keys that are considered invalid environment variable names will have those keys skipped. The Pod will be allowed to start. There will be an event whose reason is InvalidVariableNames and the message will contain the list of invalid keys that were skipped. The example shows a pod which refers to the default/mysecret that contains 2 invalid keys: 1badkey and 2alsobad.

```
kubectl get events
```

The output is similar to:

LASTSEEN	FIRSTSEEN	COUNT	NAME	KIND
SUBOBJECT			TYPE	REASON
0s	0s	1	dapi-test-pod	
Pod				Warning
			InvalidEnvironmentVariableNames	kubelet, 127.0.0.1 Keys [1badkey, 2alsobad] from the EnvFrom secret default/mysecret were skipped since they are considered invalid environment variable names.

Secret and Pod lifetime interaction

When a Pod is created by calling the Kubernetes API, there is no check if a referenced secret exists. Once a Pod is scheduled, the kubelet will try to fetch the secret value. If the secret cannot be fetched because it does not exist or because of a temporary lack of connection to the API server, the kubelet will periodically retry. It will report an event about the Pod explaining the reason it is not started yet. Once the secret is fetched, the kubelet will create and mount a volume containing it. None of the Pod's containers will start until all the Pod's volumes are mounted.

Use cases

Use-Case: As container environment variables

Create a secret

```
apiVersion: v1
kind: Secret
metadata:
  name: mysecret
type: Opaque
data:
  USER_NAME: YWRtaW4=
  PASSWORD: MlWYyZDFlMmU2N2Rm
```

Create the Secret:

```
kubectl apply -f mysecret.yaml
```

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.

```
apiVersion: v1
kind: Pod
metadata:
  name: secret-test-pod
spec:
  containers:
    - name: test-container
      image: k8s.gcr.io/busybox
      command: [ "/bin/sh", "-c", "env" ]
      envFrom:
        - secretRef:
            name: mysecret
  restartPolicy: Never
```

Use-Case: Pod with ssh keys

Create a secret containing some ssh keys:

```
kubectl create secret generic ssh-key-secret --from-file=ssh-privatekey=/path/to/.ssh/id_rsa --from-file=ssh-publickey=/path/to/.ssh/id_rsa.pub
```

The output is similar to:

```
secret "ssh-key-secret" created
```

You can also create a `kustomization.yaml` with a `secretGenerator` field containing ssh keys.

Caution: Think carefully before sending your own ssh keys: other users of the cluster may have access to the secret. Use a service account which you want to be accessible to all the users with whom you share the Kubernetes cluster, and can revoke this account if the users are compromised.

Now you can create a Pod which references the secret with the ssh key and consumes it in a volume:

```
apiVersion: v1
kind: Pod
metadata:
  name: secret-test-pod
  labels:
    name: secret-test
spec:
  volumes:
    - name: secret-volume
      secret:
        secretName: ssh-key-secret
  containers:
    - name: ssh-test-container
      image: mySshImage
      volumeMounts:
        - name: secret-volume
          readOnly: true
          mountPath: "/etc/secret-volume"
```

When the container's command runs, the pieces of the key will be available in:

```
/etc/secret-volume/ssh-publickey
/etc/secret-volume/ssh-privatekey
```

The container is then free to use the secret data to establish an ssh connection.

Use-Case: Pods with prod / test credentials

This example illustrates a Pod which consumes a secret containing production credentials and another Pod which consumes a secret with test environment credentials.

You can create a `kustomization.yaml` with a `secretGenerator` field or run `kubectl create secret`.

```
kubectl create secret generic prod-db-secret --from-literal=username=produser --from-literal=password=Y4nys7f11
```

The output is similar to:

```
secret "prod-db-secret" created
```

You can also create a secret for test environment credentials.

```
kubectl create secret generic test-db-secret --from-literal=username=testuser --from-literal=password=iluvtests
```

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 this way:

```
kubectl create secret generic dev-db-secret --from-literal=username=devuser --from-literal=password='S!B\*d$zDsb='
```

You do not need to escape special characters in passwords from files (--from-file).

Now make the Pods:

```
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
```

Add the pods to the same kustomization.yaml:

```
cat <<EOF >> kustomization.yaml  
resources:  
- pod.yaml  
EOF
```

Apply all those objects on the API server by running:

```
kubectl apply -k .
```

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
```

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.

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
```

Use-case: dotfiles in a secret volume

You can make your data "hidden" by defining a key that begins with a dot. This key represents a dotfile or "hidden" file. For example, when the following secret is mounted into a volume, `secret-volume`:

```
apiVersion: v1
kind: Secret
metadata:
  name: dotfile-secret
data:
  .secret-file: dmFsdWUtMg0KDQo=
---
apiVersion: v1
kind: Pod
metadata:
  name: secret-dotfiles-pod
spec:
  volumes:
    - name: secret-volume
      secret:
        secretName: dotfile-secret
  containers:
    - name: dotfile-test-container
      image: k8s.gcr.io/busybox
      command:
        - ls
        - "-l"
        - "/etc/secret-volume"
  volumeMounts:
    - name: secret-volume
      readOnly: true
      mountPath: "/etc/secret-volume"
```

The volume will contain a single file, called `.secret-file`, and the `dotfile-test-container` will have this file present at the path `/etc/secret-volume/.secret-file`.

Note: Files beginning with dot characters are hidden from the output of `ls -l`; you must use `ls -la` to see them when listing directory contents.

Use-case: Secret visible to one container in a Pod

Consider a program that needs to handle HTTP requests, do some complex business logic, and then sign some messages with an HMAC. Because it has

complex application logic, there might be an unnoticed remote file reading exploit in the server, which could expose the private key to an attacker.

This could be divided into two processes in two containers: a frontend container which handles user interaction and business logic, but which cannot see the private key; and a signer container that can see the private key, and responds to simple signing requests from the frontend (for example, over localhost networking).

With this partitioned approach, an attacker now has to trick the application server into doing something rather arbitrary, which may be harder than getting it to read a file.

Best practices

Clients that use the Secret API

When deploying applications that interact with the Secret API, you should limit access using [authorization policies](#) such as [RBAC](#).

Secrets often hold values that span a spectrum of importance, many of which can cause escalations within Kubernetes (e.g. service account tokens) and to external systems. Even if an individual app can reason about the power of the secrets it expects to interact with, other apps within the same namespace can render those assumptions invalid.

For these reasons `watch` and `list` requests for secrets within a namespace are extremely powerful capabilities and should be avoided, since listing secrets allows the clients to inspect the values of all secrets that are in that namespace. The ability to `watch` and `list` all secrets in a cluster should be reserved for only the most privileged, system-level components.

Applications that need to access the Secret API should perform `get` requests on the secrets they need. This lets administrators restrict access to all secrets while [white-listing access to individual instances](#) that the app needs.

For improved performance over a looping `get`, clients can design resources that reference a secret then `watch` the resource, re-requesting the secret when the reference changes. Additionally, a ["bulk watch" API](#) to let clients `watch` individual resources has also been proposed, and will likely be available in future releases of Kubernetes.

Security properties

Protections

Because secrets can be created independently of the Pods that use them, there is less risk of the secret being exposed during the workflow of creating, viewing, and editing Pods. The system can also take additional precautions with Secrets, such as avoiding writing them to disk where possible.

A secret is only sent to a node if a Pod on that node requires it. The kubelet stores the secret into a tmpfs so that the secret is not written to disk storage. Once the Pod that depends on the secret is deleted, the kubelet will delete its local copy of the secret data as well.

There may be secrets for several Pods on the same node. However, only the secrets that a Pod requests are potentially visible within its containers. Therefore, one Pod does not have access to the secrets of another Pod.

There may be several containers in a Pod. However, each container in a Pod has to request the secret volume in its volumeMounts for it to be visible within the container. This can be used to construct useful [security partitions at the Pod level](#).

On most Kubernetes distributions, communication between users and the API server, and from the API server to the kubelets, is protected by SSL/TLS. Secrets are protected when transmitted over these channels.

FEATURE STATE: Kubernetes v1.13 [beta]

You can enable [encryption at rest](#) for secret data, so that the secrets are not stored in the clear into [etcd](#).

Risks

- *In the API server, secret data is stored in [etcd](#); therefore:*
 - *Administrators should enable encryption at rest for cluster data (requires v1.13 or later).*
 - *Administrators should limit access to etcd to admin users.*
 - *Administrators may want to wipe/shred disks used by etcd when no longer in use.*
 - *If running etcd in a cluster, administrators should make sure to use SSL/TLS for etcd peer-to-peer communication.*
- *If you configure the secret through a manifest (JSON or YAML) file which has the secret data encoded as base64, sharing this file or checking it in to a source repository means the secret is compromised. Base64 encoding is not an encryption method and is considered the same as plain text.*
- *Applications still need to protect the value of secret after reading it from the volume, such as not accidentally logging it or transmitting it to an untrusted party.*
- *A user who can create a Pod that uses a secret can also see the value of that secret. Even if the API server policy does not allow that user to read the Secret, the user could run a Pod which exposes the secret.*
- *Currently, anyone with root permission on any node can read any secret from the API server, by impersonating the kubelet. It is a planned feature to only send secrets to nodes that actually require them, to restrict the impact of a root exploit on a single node.*

What's next

- Learn how to [manage Secret using kubectl](#)

- Learn how to [manage Secret using config file](#)
- Learn how to [manage Secret using kustomize](#)

Feedback

Was this page helpful?

Yes No

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

Last modified November 13, 2020 at 11:16 PM PST: [Env var are not updated after a secret update \(d77262740\)](#)

[Edit this page](#) [Create child page](#) [Create an issue](#)

- [Overview of Secrets](#)
- [Types of Secret](#)
 - [Opaque secrets](#)
 - [Service account token Secrets](#)
 - [Docker config Secrets](#)
 - [Basic authentication Secret](#)
 - [SSH authentication secrets](#)
 - [TLS secrets](#)
 - [Bootstrap token Secrets](#)
- [Creating a Secret](#)
- [Editing a Secret](#)
- [Using Secrets](#)
 - [Using Secrets as files from a Pod](#)
 - [Using Secrets as environment variables](#)
- [Immutable Secrets](#)
 - [Using imagePullSecrets](#)
 - [Arranging for imagePullSecrets to be automatically attached](#)
 - [Automatic mounting of manually created Secrets](#)
- [Details](#)
 - [Restrictions](#)
 - [Secret and Pod lifetime interaction](#)
- [Use cases](#)
 - [Use-Case: As container environment variables](#)
 - [Use-Case: Pod with ssh keys](#)
 - [Use-Case: Pods with prod / test credentials](#)
 - [Use-case: dotfiles in a secret volume](#)
 - [Use-case: Secret visible to one container in a Pod](#)
- [Best practices](#)
 - [Clients that use the Secret API](#)
- [Security properties](#)
 - [Protections](#)
 - [Risks](#)
- [What's next](#)

Managing Resources for Containers

When you specify a [Pod](#), you can optionally specify how much of each resource a [Container](#) needs. The most common resources to specify are CPU and memory (RAM); there are others.

When you specify the resource request for Containers in a Pod, the scheduler uses this information to decide which node to place the Pod on. When you specify a resource limit for a Container, the kubelet enforces those limits so that the running container is not allowed to use more of that resource than the limit you set. The kubelet also reserves at least the request amount of that system resource specifically for that container to use.

Requests and limits

If the node where a Pod is running has enough of a resource available, it's possible (and allowed) for a container to use more resource than its request for that resource specifies. However, a container is not allowed to use more than its resource limit.

For example, if you set a memory request of 256 MiB for a container, and that container is in a Pod scheduled to a Node with 8GiB of memory and no other Pods, then the container can try to use more RAM.

If you set a memory limit of 4GiB for that Container, the kubelet (and [container runtime](#)) enforce the limit. The runtime prevents the container from using more than the configured resource limit. For example: when a process in the container tries to consume more than the allowed amount of memory, the system kernel terminates the process that attempted the allocation, with an out of memory (OOM) error.

Limits can be implemented either reactively (the system intervenes once it sees a violation) or by enforcement (the system prevents the container from ever exceeding the limit). Different runtimes can have different ways to implement the same restrictions.

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.

Resource types

CPU and memory are each a resource type. A resource type has a base unit. CPU represents compute processing and is specified in units of [Kubernetes](#)

[CPUs](#). Memory is specified in units of bytes. If you're using Kubernetes v1.14 or newer, you can specify huge page resources. Huge pages are a Linux-specific feature where the node kernel allocates blocks of memory that are much larger than the default page size.

For example, on a system where the default page size is 4KiB, you could specify a limit, `hugepages-2Mi: 80Mi`. If the container tries allocating over 40 2MiB huge pages (a total of 80 MiB), that allocation fails.

Note: You cannot overcommit `hugepages-*` resources. This is different from the `memory` and `cpu` resources.

CPU and memory are collectively referred to as *compute resources*, or just *resources*. Compute resources are measurable quantities that can be requested, allocated, and consumed. They are distinct from [API resources](#). API resources, such as *Pods* and [Services](#) are objects that can be read and modified through the Kubernetes API server.

Resource requests and limits of Pod and Container

Each Container of a Pod can specify one or more of the following:

- `spec.containers[].resources.limits.cpu`
- `spec.containers[].resources.limits.memory`
- `spec.containers[].resources.limits.hugepages-<size>`
- `spec.containers[].resources.requests.cpu`
- `spec.containers[].resources.requests.memory`
- `spec.containers[].resources.requests.hugepages-<size>`

Although *requests* and *limits* can only be specified on individual Containers, it is convenient to talk about Pod resource requests and limits. A Pod resource request/limit for a particular resource type is the sum of the resource requests/limits of that type for each Container in the Pod.

Resource units in Kubernetes

Meaning of CPU

Limits and requests for CPU resources are measured in *cpu* units. One *cpu*, in Kubernetes, is equivalent to **1 vCPU/Core** for cloud providers and **1 hyperthread** on bare-metal Intel processors.

Fractional requests are allowed. A Container with `spec.containers[].resources.requests.cpu` of `0.5` is guaranteed half as much CPU as one that asks for 1 CPU. The expression `0.1` is equivalent to the expression `100m`, which can be read as "one hundred millicpu". Some people say "one hundred millicores", and this is understood to mean the same thing. A request with a decimal point, like `0.1`, is converted to `100m` by the API, and precision finer than `1m` is not allowed. For this reason, the form `100m` might be preferred.

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.

Meaning of memory

Limits and requests for memory are measured in bytes. You can express memory as a plain integer or as a fixed-point number using one of these suffixes: E, P, T, G, M, K. You can also use the power-of-two equivalents: Ei, Pi, Ti, Gi, Mi, Ki. For example, the following represent roughly the same value:

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

Here's an example. The following Pod has two Containers. Each Container has a request of 0.25 cpu and 64MiB (2²⁶ bytes) of memory. Each Container has a limit of 0.5 cpu and 128MiB of memory. You can say the Pod has a request of 0.5 cpu and 128 MiB of memory, and a limit of 1 cpu and 256MiB of memory.

```
apiVersion: v1
kind: Pod
metadata:
  name: frontend
spec:
  containers:
    - name: app
      image: images.my-company.example/app:v4
      resources:
        requests:
          memory: "64Mi"
          cpu: "250m"
        limits:
          memory: "128Mi"
          cpu: "500m"
    - name: log-aggregator
      image: images.my-company.example/log-aggregator:v6
      resources:
        requests:
          memory: "64Mi"
          cpu: "250m"
        limits:
          memory: "128Mi"
          cpu: "500m"
```

How Pods with resource requests are scheduled

When you create a Pod, the Kubernetes scheduler selects a node for the Pod to run on. Each node has a maximum capacity for each of the resource

types: the amount of CPU and memory it can provide for Pods. The scheduler ensures that, for each resource type, the sum of the resource requests of the scheduled Containers is less than the capacity of the node. Note that although actual memory or CPU resource usage on nodes is very low, the scheduler still refuses to place a Pod on a node if the capacity check fails. This protects against a resource shortage on a node when resource usage later increases, for example, during a daily peak in request rate.

How Pods with resource limits are run

When the kubelet starts a Container of a Pod, it passes the CPU and memory limits to the container runtime.

When using Docker:

- *The spec.containers[].resources.requests.cpu is converted to its core value, which is potentially fractional, and multiplied by 1024. The greater of this number or 2 is used as the value of the [--cpu-shares](#) flag in the docker run command.*
- *The spec.containers[].resources.limits.cpu is converted to its millicore value and multiplied by 100. The resulting value is the total amount of CPU time that a container can use every 100ms. A container cannot use more than its share of CPU time during this interval.*

Note: *The default quota period is 100ms. The minimum resolution of CPU quota is 1ms.*

- *The spec.containers[].resources.limits.memory is converted to an integer, and used as the value of the [--memory](#) flag in the docker run command.*

If a Container exceeds its memory limit, it might be terminated. If it is restartable, the kubelet will restart it, as with any other type of runtime failure.

If a Container exceeds its memory request, it is likely that its Pod will be evicted whenever the node runs out of memory.

A Container might or might not be allowed to exceed its CPU limit for extended periods of time. However, it will not be killed for excessive CPU usage.

To determine whether a Container cannot be scheduled or is being killed due to resource limits, see the [Troubleshooting](#) section.

Monitoring compute & memory resource usage

The resource usage of a Pod is reported as part of the Pod status.

If optional [tools for monitoring](#) are available in your cluster, then Pod resource usage can be retrieved either from the [Metrics API](#) directly or from your monitoring tools.

Local ephemeral storage

FEATURE STATE: Kubernetes v1.10 [beta]

Nodes have local ephemeral storage, backed by locally-attached writeable devices or, sometimes, by RAM. "Ephemeral" means that there is no long-term guarantee about durability.

Pods use ephemeral local storage for scratch space, caching, and for logs. The kubelet can provide scratch space to Pods using local ephemeral storage to mount [emptyDir volumes](#) into containers.

The kubelet also uses this kind of storage to hold [node-level container logs](#), container images, and the writable layers of running containers.

Caution: If a node fails, the data in its ephemeral storage can be lost.

Your applications cannot expect any performance SLAs (disk IOPS for example) from local ephemeral storage.

As a beta feature, Kubernetes lets you track, reserve and limit the amount of ephemeral local storage a Pod can consume.

Configurations for local ephemeral storage

Kubernetes supports two ways to configure local ephemeral storage on a node:

- [Single filesystem](#)
- [Two filesystems](#)

In this configuration, you place all different kinds of ephemeral local data (`emptyDir` volumes, writeable layers, container images, logs) into one filesystem. The most effective way to configure the kubelet means dedicating this filesystem to Kubernetes (kubelet) data.

The kubelet also writes [node-level container logs](#) and treats these similarly to ephemeral local storage.

The kubelet writes logs to files inside its configured log directory (`/var/log` by default); and has a base directory for other locally stored data (`/var/lib/kubelet` by default).

Typically, both `/var/lib/kubelet` and `/var/log` are on the system root filesystem, and the kubelet is designed with that layout in mind.

Your node can have as many other filesystems, not used for Kubernetes, as you like.

You have a filesystem on the node that you're using for ephemeral data that comes from running Pods: logs, and `emptyDir` volumes. You can use this filesystem for other data (for example: system logs not related to Kubernetes); it can even be the root filesystem.

The kubelet also writes [node-level container logs](#) into the first filesystem, and treats these similarly to ephemeral local storage.

You also use a separate filesystem, backed by a different logical storage device. In this configuration, the directory where you tell the kubelet to place container image layers and writeable layers is on this second filesystem.

The first filesystem does not hold any image layers or writeable layers.

Your node can have as many other filesystems, not used for Kubernetes, as you like.

The kubelet can measure how much local storage it is using. It does this provided that:

- the `LocalStorageCapacityIsolation` [feature gate](#) is enabled (the feature is on by default), and
- you have set up the node using one of the supported configurations for local ephemeral storage.

If you have a different configuration, then the kubelet does not apply resource limits for ephemeral local storage.

Note: The kubelet tracks `tmpfs` `emptyDir` volumes as container memory use, rather than as local ephemeral storage.

Setting requests and limits for local ephemeral storage

You can use `ephemeral-storage` for managing local ephemeral storage. Each Container of a Pod can specify one or more of the following:

- `spec.containers[].resources.limits.ephemeral-storage`
- `spec.containers[].resources.requests.ephemeral-storage`

Limits and requests for `ephemeral-storage` are measured in bytes. You can express storage as a plain integer or as a fixed-point number using one of these suffixes: `E`, `P`, `T`, `G`, `M`, `K`. You can also use the power-of-two equivalents: `Ei`, `Pi`, `Ti`, `Gi`, `Mi`, `Ki`. For example, the following represent roughly the same value:

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

In the following example, the Pod has two Containers. Each Container has a request of 2GiB of local ephemeral storage. Each Container has a limit of 4GiB of local ephemeral storage. Therefore, the Pod has a request of 4GiB of local ephemeral storage, and a limit of 8GiB of local ephemeral storage.

```

apiVersion: v1
kind: Pod
metadata:
  name: frontend
spec:
  containers:
    - name: app
      image: images.my-company.example/app:v4
      resources:
        requests:
          ephemeral-storage: "2Gi"
        limits:
          ephemeral-storage: "4Gi"
    - name: log-aggregator
      image: images.my-company.example/log-aggregator:v6
      resources:
        requests:
          ephemeral-storage: "2Gi"
        limits:
          ephemeral-storage: "4Gi"

```

How Pods with ephemeral-storage requests are scheduled

When you create a Pod, the Kubernetes scheduler selects a node for the Pod to run on. Each node has a maximum amount of local ephemeral storage it can provide for Pods. For more information, see [Node Allocatable](#).

The scheduler ensures that the sum of the resource requests of the scheduled Containers is less than the capacity of the node.

Ephemeral storage consumption management

If the kubelet is managing local ephemeral storage as a resource, then the kubelet measures storage use in:

- *emptyDir volumes, except tmpfs emptyDir volumes*
- *directories holding node-level logs*
- *writeable container layers*

If a Pod is using more ephemeral storage than you allow it to, the kubelet sets an eviction signal that triggers Pod eviction.

For container-level isolation, if a Container's writable layer and log usage exceeds its storage limit, the kubelet marks the Pod for eviction.

For pod-level isolation the kubelet works out an overall Pod storage limit by summing the limits for the containers in that Pod. In this case, if the sum of the local ephemeral storage usage from all containers and also the Pod's emptyDir volumes exceeds the overall Pod storage limit, then the kubelet also marks the Pod for eviction.

Caution:

If the kubelet is not measuring local ephemeral storage, then a Pod that exceeds its local storage limit will not be evicted for breaching local storage resource limits.

However, if the filesystem space for writeable container layers, node-level logs, or `emptyDir` volumes falls low, the node [taints](#) itself as short on local storage and this taint triggers eviction for any Pods that don't specifically tolerate the taint.

See the supported [configurations](#) for ephemeral local storage.

The kubelet supports different ways to measure Pod storage use:

- [Periodic scanning](#)
- [Filesystem project quota](#)

The kubelet performs regular, scheduled checks that scan each `emptyDir` volume, container log directory, and writeable container layer.

The scan measures how much space is used.

Note:

In this mode, the kubelet does not track open file descriptors for deleted files.

If you (or a container) create a file inside an `emptyDir` volume, something then opens that file, and you delete the file while it is still open, then the inode for the deleted file stays until you close that file but the kubelet does not categorize the space as in use.

FEATURE STATE: Kubernetes v1.15 [alpha]

Project quotas are an operating-system level feature for managing storage use on filesystems. With Kubernetes, you can enable project quotas for monitoring storage use. Make sure that the filesystem backing the `emptyDir` volumes, on the node, provides project quota support. For example, XFS and ext4fs offer project quotas.

Note: Project quotas let you monitor storage use; they do not enforce limits.

Kubernetes uses project IDs starting from 1048576. The IDs in use are registered in `/etc/projects` and `/etc/projid`. If project IDs in this range are used for other purposes on the system, those project IDs must be registered in `/etc/projects` and `/etc/projid` so that Kubernetes does not use them.

Quotas are faster and more accurate than directory scanning. When a directory is assigned to a project, all files created under a directory are created in that project, and the kernel merely has to keep track of how many blocks are in use by files in that project.

If a file is created and deleted, but has an open file descriptor, it continues to consume space. Quota tracking records that space accurately whereas directory scans overlook the storage used by deleted files.

If you want to use project quotas, you should:

- *Enable the LocalStorageCapacityIsolationFSQuotaMonitoring=true feature gate in the kubelet configuration.*

- *Ensure that the root filesystem (or optional runtime filesystem) has project quotas enabled. All XFS filesystems support project quotas. For ext4 filesystems, you need to enable the project quota tracking feature while the filesystem is not mounted.*

```
# For ext4, with /dev/block-device not mounted
sudo tune2fs -O project -Q prjquota /dev/block-device
```

- *Ensure that the root filesystem (or optional runtime filesystem) is mounted with project quotas enabled. For both XFS and ext4fs, the mount option is named prjquota.*

Extended resources

Extended resources are fully-qualified resource names outside the kubernetes.io domain. They allow cluster operators to advertise and users to consume the non-Kubernetes-built-in resources.

There are two steps required to use Extended Resources. First, the cluster operator must advertise an Extended Resource. Second, users must request the Extended Resource in Pods.

Managing extended resources

Node-level extended resources

Node-level extended resources are tied to nodes.

Device plugin managed resources

See [Device Plugin](#) for how to advertise device plugin managed resources on each node.

Other resources

To advertise a new node-level extended resource, the cluster operator can submit a PATCH HTTP request to the API server to specify the available quantity in the status.capacity for a node in the cluster. After this

operation, the node's `status.capacity` will include a new resource. The `status.allocatable` field is updated automatically with the new resource asynchronously by the kubelet. Note that because the scheduler uses the `node status.allocatable` value when evaluating Pod fitness, there may be a short delay between patching the node capacity with a new resource and the first Pod that requests the resource to be scheduled on that node.

Example:

Here is an example showing how to use curl to form an HTTP request that advertises five "example.com/foo" resources on node `k8s-node-1` whose master is `k8s-master`.

```
curl --header "Content-Type: application/json-patch+json" \
--request PATCH \
--data '[{"op": "add", "path": "/status/capacity/
example.com~1foo", "value": "5"}]' \
http://k8s-master:8080/api/v1/nodes/k8s-node-1/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](#).

Cluster-level extended resources

Cluster-level extended resources are not tied to nodes. They are usually managed by scheduler extenders, which handle the resource consumption and resource quota.

You can specify the extended resources that are handled by scheduler extenders in [scheduler policy configuration](#).

Example:

The following configuration for a scheduler policy indicates that the cluster-level extended resource "example.com/foo" is handled by the scheduler extender.

- The scheduler sends a Pod to the scheduler extender only if the Pod requests "example.com/foo".
- The `ignoredByScheduler` field specifies that the scheduler does not check the "example.com/foo" resource in its `PodFitsResources` predicate.

```
{
  "kind": "Policy",
  "apiVersion": "v1",
  "extenders": [
    {
      "urlPrefix": "<extender-endpoint>",
      "bindVerb": "bind",
      "managedResources": [
```

```

    {
      "name": "example.com/foo",
      "ignoredByScheduler": true
    }
  ]
}

```

Consuming extended resources

Users can consume extended resources in Pod specs just like CPU and memory. The scheduler takes care of the resource accounting so that no more than the available amount is simultaneously allocated to Pods.

The API server restricts quantities of extended resources to whole numbers. Examples of valid quantities are 3, 3000m and 3Ki. Examples of invalid quantities are 0.5 and 1500m.

Note: Extended resources replace Opaque Integer Resources.
Users can use any domain name prefix other than kubernetes.io which is reserved.

To consume an extended resource in a Pod, include the resource name as a key in the spec.containers[].resources.limits map in the container spec.

Note: Extended resources cannot be overcommitted, so request and limit must be equal if both are present in a container spec.

A Pod is scheduled only if all of the resource requests are satisfied, including CPU, memory and any extended resources. The Pod remains in the PENDING state as long as the resource request cannot be satisfied.

Example:

The Pod below requests 2 CPUs and 1 "example.com/foo" (an extended resource).

```

apiVersion: v1
kind: Pod
metadata:
  name: my-pod
spec:
  containers:
  - name: my-container
    image: myimage
    resources:
      requests:
        cpu: 2
        example.com/foo: 1
      limits:
        example.com/foo: 1

```

PID limiting

Process ID (PID) limits allow for the configuration of a kubelet to limit the number of PIDs that a given Pod can consume. See [Pid Limiting](#) for information.

Troubleshooting

My Pods are pending with event message failedScheduling

If the scheduler cannot find any node where a Pod can fit, the Pod remains unscheduled until a place can be found. An event is produced each time the scheduler fails to find a place for the Pod, like this:

```
kubectl describe pod frontend | grep -A 3 Events
```

```
Events:
FirstSeen LastSeen   Count  From           Subobject
PathReason      Message
 36s    5s       6      {scheduler}
FailedScheduling Failed for reason PodExceedsFreeCPU and
possibly others
```

In the preceding example, the Pod named "frontend" fails to be scheduled due to insufficient CPU resource on the node. Similar error messages can also suggest failure due to insufficient memory (PodExceedsFreeMemory). In general, if a Pod is pending with a message of this type, there are several things to try:

- Add more nodes to the cluster.
- Terminate unneeded Pods to make room for pending Pods.
- Check that the Pod is not larger than all the nodes. For example, if all the nodes have a capacity of `cpu: 1`, then a Pod with a request of `cpu: 1.1` will never be scheduled.

You can check node capacities and amounts allocated with the `kubectl describe nodes` command. For example:

```
kubectl describe nodes e2e-test-node-pool-4lw4
```

```
Name:           e2e-test-node-pool-4lw4
[ ... lines removed for clarity ... ]
Capacity:
  cpu:          2
  memory:       7679792Ki
  pods:         110
Allocatable:
  cpu:          1800m
  memory:       7474992Ki
  pods:         110
[ ... lines removed for clarity ... ]
```

Non-terminated Pods:			(5 in total)		
Namespace	Name		CPU		
Requests	CPU	Limits	Memory Requests	Memory	Limits
<hr/>					
kube-system	fluentd-gcp-v1.38-28bv1				100m
(5%)	0 (0%)	200Mi (2%)	200Mi (2%)		
kube-system	kube-dns-3297075139-61lj3				260m
(13%)	0 (0%)	100Mi (1%)	170Mi (2%)		
kube-system	kube-proxy-e2e-test-...				100m
(5%)	0 (0%)	0 (0%)	0 (0%)		
kube-system	monitoring-influxdb-grafana-v4-z1m12				200m
(10%)	200m (10%)	600Mi (8%)	600Mi (8%)		
kube-system	node-problem-detector-v0.1-fj7m3				20m
(1%)	200m (10%)	20Mi (0%)	100Mi (1%)		
Allocated resources:					
(Total limits may be over 100 percent, i.e., overcommitted.)					
CPU Requests	CPU Limits		Memory Requests	Memory	Limits
<hr/>	<hr/>	<hr/>	<hr/>	<hr/>	<hr/>
680m (34%)	400m (20%)		920Mi (11%)		1070Mi (13%)

In the preceding output, you can see that if a Pod requests more than 1120m CPUs or 6.23Gi of memory, it will not fit on the node.

By looking at the *Pods* section, you can see which Pods are taking up space on the node.

The amount of resources available to Pods is less than the node capacity, because system daemons use a portion of the available resources. The allocatable field [NodeStatus](#) gives the amount of resources that are available to Pods. For more information, see [Node Allocatable Resources](#).

The [resource quota](#) feature can be configured to limit the total amount of resources that can be consumed. If used in conjunction with namespaces, it can prevent one team from hogging all the resources.

My Container is terminated

Your Container might get terminated because it is resource-starved. To check whether a Container is being killed because it is hitting a resource limit, call `kubectl describe pod` on the Pod of interest:

```
kubectl describe pod simmemleak-hra99
```

Name:	simmemleak-hra99
Namespace:	default
Image(s):	saadali/simmemleak
Node:	kubernetes-node-tf0f/
10.240.216.66	
Labels:	name=simmemleak
Status:	Running
Reason:	

```

Message:
IP: 10.244.2.75
Replication Controllers: simmemleak (1/1 replicas created)
Containers:
  simmemleak:
    Image: saadali/simmemleak
    Limits:
      cpu: 100m
      memory: 50Mi
    State:
      Started: Tue, 07 Jul 2015 12:54:41 -0700
    Last Termination State: Terminated
      Exit Code: 1
      Started: Fri, 07 Jul 2015 12:54:30 -0700
      Finished: Fri, 07 Jul 2015 12:54:33 -0700
    Ready: False
    Restart Count: 5
Conditions:
  Type Status
  Ready False
Events:
  FirstSeen          LastSeen        Count
  From
SubobjectPath           Reason       Message
  Tue, 07 Jul 2015 12:53:51 -0700   Tue, 07 Jul 2015 12:53:51
-0700 1
{scheduler }           scheduled   Successfully assigned simmemleak-hra99 to
kubernetes-node-tf0f
  Tue, 07 Jul 2015 12:53:51 -0700   Tue, 07 Jul 2015 12:53:51
-0700 1           {kubelet kubernetes-node-tf0f} implicitly
required container POD pulled Pod container image
"k8s.gcr.io/pause:0.8.0" already present on machine
  Tue, 07 Jul 2015 12:53:51 -0700   Tue, 07 Jul 2015 12:53:51
-0700 1           {kubelet kubernetes-node-tf0f} implicitly
required container POD created Created with docker id
6a41280f516d
  Tue, 07 Jul 2015 12:53:51 -0700   Tue, 07 Jul 2015 12:53:51
-0700 1           {kubelet kubernetes-node-tf0f} implicitly
required container POD started Started with docker id
6a41280f516d
  Tue, 07 Jul 2015 12:53:51 -0700   Tue, 07 Jul 2015 12:53:51
-0700 1           {kubelet kubernetes-node-tf0f}
spec.containers{simmemleak}           created     Created with
docker id 87348f12526a

```

In the preceding example, the `Restart Count: 5` indicates that the `simmemleak` Container in the Pod was terminated and restarted five times.

You can call `kubectl get pod` with the `-o go-template=...` option to fetch the status of previously terminated Containers:

```
kubectl get pod -o go-template='{{range.status.containerStatuses}}  
{{"Container Name: "}}{{.name}}{{$lastState: "\r\nLastState: "}}  
{{.lastState}}{{end}}' simmemleak-hra99
```

```
Container Name: simmemleak  
LastState: map[terminated:map[exitCode:137 reason:OOM Killed  
startedAt:2015-07-07T20:58:43Z finishedAt:2015-07-07T20:58:43Z  
containerID:docker://  
0e4095bba1feccdfc7ef9fb6ebffe972b4b14285d5acdec6f0d3ae8a22fad8b2]  
]
```

You can see that the Container was terminated because of `reason:OOM Killed`, where `OOM` stands for Out Of Memory.

What's next

- Get hands-on experience [assigning Memory resources to Containers and Pods](#).
- Get hands-on experience [assigning CPU resources to Containers and Pods](#).
- For more details about the difference between requests and limits, see [Resource QoS](#).
- Read the [Container API reference](#)
- Read the [ResourceRequirements API reference](#)
- Read about [project quotas](#) in XFS

Feedback

Was this page helpful?

Yes No

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

Last modified December 17, 2020 at 8:43 PM PST: [Fix typo in manage-resources-containers.md \(815fe3790\)](#)

[Edit this page](#) [Create child page](#) [Create an issue](#)

- [Requests and limits](#)
- [Resource types](#)
- [Resource requests and limits of Pod and Container](#)
- [Resource units in Kubernetes](#)
 - [Meaning of CPU](#)
 - [Meaning of memory](#)
- [How Pods with resource requests are scheduled](#)

- [How Pods with resource limits are run](#)
 - [Monitoring compute & memory resource usage](#)
- [Local ephemeral storage](#)
 - [Configurations for local ephemeral storage](#)
 - [Setting requests and limits for local ephemeral storage](#)
 - [How Pods with ephemeral-storage requests are scheduled](#)
 - [Ephemeral storage consumption management](#)
- [Extended resources](#)
 - [Managing extended resources](#)
 - [Consuming extended resources](#)
- [PID limiting](#)
- [Troubleshooting](#)
 - [My Pods are pending with event message failedScheduling](#)
 - [My Container is terminated](#)
- [What's next](#)

Organizing Cluster Access Using kubeconfig Files

Use kubeconfig files to organize information about clusters, users, namespaces, and authentication mechanisms. The kubectl command-line tool uses kubeconfig files to find the information it needs to choose a cluster and communicate with the API server of a cluster.

Note: A file that is used to configure access to clusters is 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*.

By default, kubectl looks for a file named config in the \$HOME/.kube directory. You can specify other kubeconfig files by setting the KUBECONFIG environment variable or by setting the [--kubeconfig](#) flag.

For step-by-step instructions on creating and specifying kubeconfig files, see [Configure Access to Multiple Clusters](#).

Supporting multiple clusters, users, and authentication mechanisms

Suppose you have several clusters, and your users and components authenticate in a variety of ways. For example:

- A running kubelet might authenticate using certificates.
- A user might authenticate using tokens.
- Administrators might have sets of certificates that they provide to individual users.

With kubeconfig files, you can organize your clusters, users, and namespaces. You can also define contexts to quickly and easily switch between clusters and namespaces.

Context

A context element in a kubeconfig file is used to group access parameters under a convenient name. Each context has three parameters: cluster, namespace, and user. By default, the kubectl command-line tool uses parameters from the current context to communicate with the cluster.

To choose the current context:

```
kubectl config use-context
```

The KUBECONFIG environment variable

The KUBECONFIG environment variable holds a list of kubeconfig files. For Linux and Mac, the list is colon-delimited. For Windows, the list is semicolon-delimited. The KUBECONFIG environment variable is not required. If the KUBECONFIG environment variable doesn't exist, kubectl uses the default kubeconfig file, \$HOME/.kube/config.

If the KUBECONFIG environment variable does exist, kubectl uses an effective configuration that is the result of merging the files listed in the KUBECONFIG environment variable.

Merging kubeconfig files

To see your configuration, enter this command:

```
kubectl config view
```

As described previously, the output might be from a single kubeconfig file, or it might be the result of merging several kubeconfig files.

Here are the rules that kubectl uses when it merges kubeconfig files:

1. If the --kubeconfig flag is set, use only the specified file. Do not merge. Only one instance of this flag is allowed.

Otherwise, if the KUBECONFIG environment variable is set, use it as a list of files that should be merged. Merge the files listed in the KUBECONFIG environment variable according to these rules:

- Ignore empty filenames.
- Produce errors for files with content that cannot be deserialized.
- The first file to set a particular value or map key wins.
- Never change the value or map key. Example: Preserve the context of the first file to set current-context. Example: If two files specify a red-user, use only values from the first file's red-user. Even if the second file has non-conflicting entries under red-user, discard them.

For an example of setting the KUBECONFIG environment variable, see [Setting the KUBECONFIG environment variable](#).

Otherwise, use the default kubeconfig file, \$HOME/.kube/config, with no merging.

2. Determine the context to use based on the first hit in this chain:

- 1. Use the --context command-line flag if it exists.*
- 2. Use the current-context from the merged kubeconfig files.*

An empty context is allowed at this point.

3. Determine the cluster and user. At this point, there might or might not be a context. Determine the cluster and user based on the first hit in this chain, which is run twice: once for user and once for cluster:

- 1. Use a command-line flag if it exists: --user or --cluster.*
- 2. If the context is non-empty, take the user or cluster from the context.*

The user and cluster can be empty at this point.

4. Determine the actual cluster information to use. At this point, there might or might not be cluster information. Build each piece of the cluster information based on this chain; the first hit wins:

- 1. Use command line flags if they exist: --server, --certificate-authority, --insecure-skip-tls-verify.*
- 2. If any cluster information attributes exist from the merged kubeconfig files, use them.*
- 3. If there is no server location, fail.*

5. Determine the actual user information to use. Build user information using the same rules as cluster information, except allow only one authentication technique per user:

- 1. Use command line flags if they exist: --client-certificate, --client-key, --username, --password, --token.*
- 2. Use the user fields from the merged kubeconfig files.*
- 3. If there are two conflicting techniques, fail.*

6. For any information still missing, use default values and potentially prompt for authentication information.

File references

File and path references in a kubeconfig file are relative to the location of the kubeconfig file. File references on the command line are relative to the current working directory. In \$HOME/.kube/config, relative paths are stored relatively, and absolute paths are stored absolutely.

What's next

- [Configure Access to Multiple Clusters](#)
- [kubectl config](#)

Feedback

Was this page helpful?

Yes No

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

Last modified May 30, 2020 at 3:10 PM PST: [add en pages \(ecc27bbbe\)](#)
[Edit this page](#) [Create child page](#) [Create an issue](#)

- [Supporting multiple clusters, users, and authentication mechanisms](#)
- [Context](#)
- [The KUBECONFIG environment variable](#)
- [Merging kubeconfig files](#)
- [File references](#)
- [What's next](#)

Pod Priority and Preemption

FEATURE STATE: Kubernetes v1.14 [stable]

[Pods](#) can have priority. Priority indicates the importance of a Pod relative to other Pods. If a Pod cannot be scheduled, the scheduler tries to preempt (evict) lower priority Pods to make scheduling of the pending Pod possible.

Warning:

In a cluster where not all users are trusted, a malicious user could create Pods at the highest possible priorities, causing other Pods to be evicted/not get scheduled. An administrator can use ResourceQuota to prevent users from creating pods at high priorities.

See [limit Priority Class consumption by default](#) for details.

How to use priority and preemption

To use priority and preemption:

1. Add one or more [PriorityClasses](#).

- Create Pods with [`priorityClassName`](#) set to one of the added PriorityClasses. Of course you do not need to create the Pods directly; normally you would add `priorityClassName` to the Pod template of a collection object like a Deployment.
2. PriorityClasses. Of course you do not need to create the Pods directly; normally you would add `priorityClassName` to the Pod template of a collection object like a Deployment.

Keep reading for more information about these steps.

Note: Kubernetes already ships with two PriorityClasses: `system-cluster-critical` and `system-node-critical`. These are common classes and are used to [ensure that critical components are always scheduled first](#).

PriorityClass

A `PriorityClass` is a non-namespaced object that defines a mapping from a priority class name to the integer value of the priority. The name is specified in the `name` field of the `PriorityClass` object's metadata. The value is specified in the `requiredValue` field. The higher the value, the higher the priority. The name of a `PriorityClass` object must be a valid [DNS subdomain name](#), and it cannot be prefixed with `system-`.

A `PriorityClass` object can have any 32-bit integer value smaller than or equal to 1 billion. Larger numbers are reserved for critical system Pods that should not normally be preempted or evicted. A cluster admin should create one `PriorityClass` object for each such mapping that they want.

`PriorityClass` also has two optional fields: `globalDefault` and `description`. The `globalDefault` field indicates that the value of this `PriorityClass` should be used for Pods without a `priorityClassName`. Only one `PriorityClass` with `globalDefault` set to true can exist in the system. If there is no `PriorityClass` with `globalDefault` set, the priority of Pods with no `priorityClassName` is zero.

The `description` field is an arbitrary string. It is meant to tell users of the cluster when they should use this `PriorityClass`.

Notes about PodPriority and existing clusters

- If you upgrade an existing cluster without this feature, the priority of your existing Pods is effectively zero.
- Addition of a `PriorityClass` with `globalDefault` set to true does not change the priorities of existing Pods. The value of such a `PriorityClass` is used only for Pods created after the `PriorityClass` is added.
- If you delete a `PriorityClass`, existing Pods that use the name of the deleted `PriorityClass` remain unchanged, but you cannot create more Pods that use the name of the deleted `PriorityClass`.

Example PriorityClass

```
apiVersion: scheduling.k8s.io/v1
kind: PriorityClass
metadata:
  name: high-priority
  value: 1000000
globalDefault: false
description: "This priority class should be used for XYZ service
pods only."
```

Non-preempting PriorityClass

FEATURE STATE: Kubernetes v1.19 [beta]

Pods with PreemptionPolicy: Never will be placed in the scheduling queue ahead of lower-priority pods, but they cannot preempt other pods. A non-preempting pod waiting to be scheduled will stay in the scheduling queue, until sufficient resources are free, and it can be scheduled. Non-preempting pods, like other pods, are subject to scheduler back-off. This means that if the scheduler tries these pods and they cannot be scheduled, they will be retried with lower frequency, allowing other pods with lower priority to be scheduled before them.

Non-preempting pods may still be preempted by other, high-priority pods.

PreemptionPolicy defaults to PreemptLowerPriority, which will allow pods of that PriorityClass to preempt lower-priority pods (as is existing default behavior). If PreemptionPolicy is set to Never, pods in that PriorityClass will be non-preempting.

An example use case is for data science workloads. A user may submit a job that they want to be prioritized above other workloads, but do not wish to discard existing work by preempting running pods. The high priority job with PreemptionPolicy: Never will be scheduled ahead of other queued pods, as soon as sufficient cluster resources "naturally" become free.

Example Non-preempting PriorityClass

```
apiVersion: scheduling.k8s.io/v1
kind: PriorityClass
metadata:
  name: high-priority-nonpreempting
  value: 1000000
  preemptionPolicy: Never
  globalDefault: false
description: "This priority class will not cause other pods to
be preempted."
```

Pod priority

After you have one or more PriorityClasses, you can create Pods that specify one of those PriorityClass names in their specifications. The priority admission controller uses the `priorityClassName` field and populates the integer value of the priority. If the priority class is not found, the Pod is rejected.

The following YAML is an example of a Pod configuration that uses the PriorityClass created in the preceding example. The priority admission controller checks the specification and resolves the priority of the Pod to 1000000.

```
apiVersion: v1
kind: Pod
metadata:
  name: nginx
  labels:
    env: test
spec:
  containers:
  - name: nginx
    image: nginx
    imagePullPolicy: IfNotPresent
    priorityClassName: high-priority
```

Effect of Pod priority on scheduling order

When Pod priority is enabled, the scheduler orders pending Pods by their priority and a pending Pod is placed ahead of other pending Pods with lower priority in the scheduling queue. As a result, the higher priority Pod may be scheduled sooner than Pods with lower priority if its scheduling requirements are met. If such Pod cannot be scheduled, scheduler will continue and tries to schedule other lower priority Pods.

Preemption

When Pods are created, they go to a queue and wait to be scheduled. The scheduler picks a Pod from the queue and tries to schedule it on a Node. If no Node is found that satisfies all the specified requirements of the Pod, preemption logic is triggered for the pending Pod. Let's call the pending Pod `P`. Preemption logic tries to find a Node where removal of one or more Pods with lower priority than `P` would enable `P` to be scheduled on that Node. If such a Node is found, one or more lower priority Pods get evicted from the Node. After the Pods are gone, `P` can be scheduled on the Node.

User exposed information

When Pod `P` preempts one or more Pods on Node `N`, `nominateNodeName` field of Pod `P`'s status is set to the name of Node `N`. This field helps

scheduler track resources reserved for Pod P and also gives users information about preemptions in their clusters.

Please note that Pod P is not necessarily scheduled to the "nominated Node". After victim Pods are preempted, they get their graceful termination period. If another node becomes available while scheduler is waiting for the victim Pods to terminate, scheduler will use the other node to schedule Pod P. As a result `nominatedNodeName` and `nodeName` of Pod spec are not always the same. Also, if scheduler preempts Pods on Node N, but then a higher priority Pod than Pod P arrives, scheduler may give Node N to the new higher priority Pod. In such a case, scheduler clears `nominatedNodeName` of Pod P. By doing this, scheduler makes Pod P eligible to preempt Pods on another Node.

Limitations of preemption

Graceful termination of preemption victims

When Pods are preempted, the victims get their [graceful termination period](#). They have that much time to finish their work and exit. If they don't, they are killed. This graceful termination period creates a time gap between the point that the scheduler preempts Pods and the time when the pending Pod (P) can be scheduled on the Node (N). In the meantime, the scheduler keeps scheduling other pending Pods. As victims exit or get terminated, the scheduler tries to schedule Pods in the pending queue. Therefore, there is usually a time gap between the point that scheduler preempts victims and the time that Pod P is scheduled. In order to minimize this gap, one can set graceful termination period of lower priority Pods to zero or a small number.

PodDisruptionBudget is supported, but not guaranteed

A [PodDisruptionBudget](#) (PDB) allows application owners to limit the number of Pods of a replicated application that are down simultaneously from voluntary disruptions. Kubernetes supports PDB when preempting Pods, but respecting PDB is best effort. The scheduler tries to find victims whose PDB are not violated by preemption, but if no such victims are found, preemption will still happen, and lower priority Pods will be removed despite their PDBs being violated.

Inter-Pod affinity on lower-priority Pods

A Node is considered for preemption only when the answer to this question is yes: "If all the Pods with lower priority than the pending Pod are removed from the Node, can the pending Pod be scheduled on the Node?"

Note: Preemption does not necessarily remove all lower-priority Pods. If the pending Pod can be scheduled by removing fewer than all lower-priority Pods, then only a portion of the lower-priority Pods are removed. Even so, the answer to the preceding question must be yes. If the answer is no, the Node is not considered for preemption.

If a pending Pod has inter-pod affinity to one or more of the lower-priority Pods on the Node, the inter-Pod affinity rule cannot be satisfied in the absence of those lower-priority Pods. In this case, the scheduler does not preempt any Pods on the Node. Instead, it looks for another Node. The scheduler might find a suitable Node or it might not. There is no guarantee that the pending Pod can be scheduled.

Our recommended solution for this problem is to create inter-Pod affinity only towards equal or higher priority Pods.

Cross node preemption

Suppose a Node N is being considered for preemption so that a pending Pod P can be scheduled on N. P might become feasible on N only if a Pod on another Node is preempted. Here's an example:

- Pod P is being considered for Node N.
- Pod Q is running on another Node in the same Zone as Node N.
- Pod P has Zone-wide anti-affinity with Pod Q (`topologyKey: topology.kubernetes.io/zone`).
- There are no other cases of anti-affinity between Pod P and other Pods in the Zone.
- In order to schedule Pod P on Node N, Pod Q can be preempted, but scheduler does not perform cross-node preemption. So, Pod P will be deemed unschedulable on Node N.

If Pod Q were removed from its Node, the Pod anti-affinity violation would be gone, and Pod P could possibly be scheduled on Node N.

We may consider adding cross Node preemption in future versions if there is enough demand and if we find an algorithm with reasonable performance.

Troubleshooting

Pod priority and pre-emption can have unwanted side effects. Here are some examples of potential problems and ways to deal with them.

Pods are preempted unnecessarily

Preemption removes existing Pods from a cluster under resource pressure to make room for higher priority pending Pods. If you give high priorities to certain Pods by mistake, these unintentionally high priority Pods may cause preemption in your cluster. Pod priority is specified by setting the `priorityClassName` field in the Pod's specification. The integer value for priority is then resolved and populated to the `priority` field of `podSpec`.

To address the problem, you can change the `priorityClassName` for those Pods to use lower priority classes, or leave that field empty. An empty `priorityClassName` is resolved to zero by default.

When a Pod is preempted, there will be events recorded for the preempted Pod. Preemption should happen only when a cluster does not have enough resources for a Pod. In such cases, preemption happens only when the priority of the pending Pod (preemptor) is higher than the victim Pods. Preemption must not happen when there is no pending Pod, or when the pending Pods have equal or lower priority than the victims. If preemption happens in such scenarios, please file an issue.

Pods are preempted, but the preemptor is not scheduled

When pods are preempted, they receive their requested graceful termination period, which is by default 30 seconds. If the victim Pods do not terminate within this period, they are forcibly terminated. Once all the victims go away, the preemptor Pod can be scheduled.

While the preemptor Pod is waiting for the victims to go away, a higher priority Pod may be created that fits on the same Node. In this case, the scheduler will schedule the higher priority Pod instead of the preemptor.

This is expected behavior: the Pod with the higher priority should take the place of a Pod with a lower priority.

Higher priority Pods are preempted before lower priority pods

The scheduler tries to find nodes that can run a pending Pod. If no node is found, the scheduler tries to remove Pods with lower priority from an arbitrary node in order to make room for the pending pod. If a node with low priority Pods is not feasible to run the pending Pod, the scheduler may choose another node with higher priority Pods (compared to the Pods on the other node) for preemption. The victims must still have lower priority than the preemptor Pod.

When there are multiple nodes available for preemption, the scheduler tries to choose the node with a set of Pods with lowest priority. However, if such Pods have PodDisruptionBudget that would be violated if they are preempted then the scheduler may choose another node with higher priority Pods.

When multiple nodes exist for preemption and none of the above scenarios apply, the scheduler chooses a node with the lowest priority.

Interactions between Pod priority and quality of service

Pod priority and [QoS class](#) are two orthogonal features with few interactions and no default restrictions on setting the priority of a Pod based on its QoS classes. The scheduler's preemption logic does not consider QoS when choosing preemption targets. Preemption considers Pod priority and attempts to choose a set of targets with the lowest priority. Higher-priority Pods are considered for preemption only if the removal of the lowest priority

Pods is not sufficient to allow the scheduler to schedule the preemptor Pod, or if the lowest priority Pods are protected by *PodDisruptionBudget*.

The only component that considers both QoS and Pod priority is [kubelet out-of-resource eviction](#). The kubelet ranks Pods for eviction first by whether or not their usage of the starved resource exceeds requests, then by Priority, and then by the consumption of the starved compute resource relative to the Pods' scheduling requests. See [evicting end-user pods](#) for more details.

kubelet out-of-resource eviction does not evict Pods when their usage does not exceed their requests. If a Pod with lower priority is not exceeding its requests, it won't be evicted. Another Pod with higher priority that exceeds its requests may be evicted.

What's next

- Read about using ResourceQuotas in connection with PriorityClasses: [limit Priority Class consumption by default](#)

Feedback

Was this page helpful?

Yes No

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

Last modified October 07, 2020 at 7:16 PM PST: [Revise cluster management task \(59dcd57cc\)](#)

[Edit this page](#) [Create child page](#) [Create an issue](#)

- [How to use priority and preemption](#)
- [PriorityClass](#)
 - [Notes about PodPriority and existing clusters](#)
 - [Example PriorityClass](#)
- [Non-preempting PriorityClass](#)
 - [Example Non-preempting PriorityClass](#)
- [Pod priority](#)
 - [Effect of Pod priority on scheduling order](#)
- [Preemption](#)
 - [User exposed information](#)
 - [Limitations of preemption](#)
- [Troubleshooting](#)
 - [Pods are preempted unnecessarily](#)
 - [Pods are preempted, but the preemptor is not scheduled](#)
 - [Higher priority Pods are preempted before lower priority pods](#)
- [Interactions between Pod priority and quality of service](#)
- [What's next](#)

Security

Concepts for keeping your cloud-native workload secure.

[Overview of Cloud Native Security](#)

[Pod Security Standards](#)

[Controlling Access to the Kubernetes API](#)

Overview of Cloud Native Security

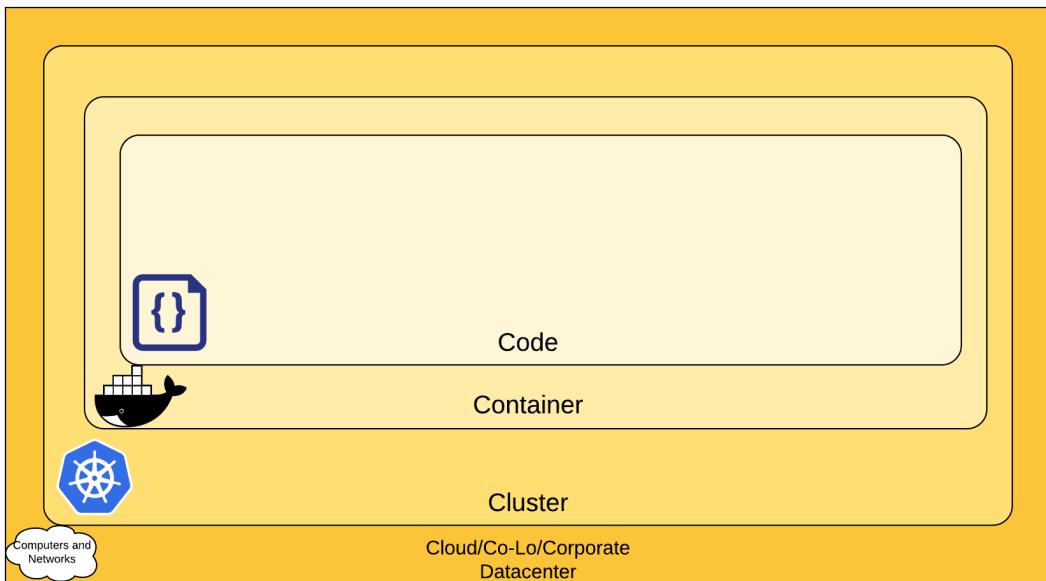
This overview defines a model for thinking about Kubernetes security in the context of Cloud Native security.

Warning: This container security model provides suggestions, not proven information security policies.

The 4C's of Cloud Native security

You can think about security in layers. The 4C's of Cloud Native security are *Cloud, Clusters, Containers, and Code*.

Note: This layered approach augments the [defense in depth](#) computing approach to security, which is widely regarded as a best practice for securing software systems.



The 4C's of Cloud Native Security

Each layer of the Cloud Native security model builds upon the next outermost layer. The Code layer benefits from strong base (Cloud, Cluster, Container) security layers. You cannot safeguard against poor security standards in the base layers by addressing security at the Code level.

Cloud

In many ways, the Cloud (or co-located servers, or the corporate datacenter) is the [trusted computing base](#) of a Kubernetes cluster. If the Cloud layer is vulnerable (or configured in a vulnerable way) then there is no guarantee that the components built on top of this base are secure. Each cloud provider makes security recommendations for running workloads securely in their environment.

Cloud provider security

If you are running a Kubernetes cluster on your own hardware or a different cloud provider, consult your documentation for security best practices. Here are links to some of the popular cloud providers' security documentation:

IaaS Provider	Link
Alibaba Cloud	https://www.alibabacloud.com/trust-center
Amazon Web Services	https://aws.amazon.com/security/
Google Cloud Platform	https://cloud.google.com/security/
IBM Cloud	https://www.ibm.com/cloud/security
Microsoft Azure	https://docs.microsoft.com/en-us/azure/security/azure-security
VMWare VSphere	https://www.vmware.com/security/hardening-guides.html

Infrastructure security

Suggestions for securing your infrastructure in a Kubernetes cluster:

Area of Concern for Kubernetes Infrastructure	Recommendation
Network access to API Server (Control plane)	All access to the Kubernetes control plane is not allowed publicly on the internet and is controlled by network access control lists restricted to the set of IP addresses needed to administer the cluster.
Network access to Nodes (nodes)	Nodes should be configured to <i>only</i> accept connections (via network access control lists) from the control plane on the specified ports, and accept connections for services in Kubernetes of type NodePort and LoadBalancer. If possible, these nodes should not be exposed on the public internet entirely.

Area of Concern for Kubernetes Infrastructure	Recommendation
Kubernetes access to Cloud Provider API	Each cloud provider needs to grant a different set of permissions to the Kubernetes control plane and nodes. It is best to provide the cluster with cloud provider access that follows the principle of least privilege for the resources it needs to administer. The Kops documentation provides information about IAM policies and roles.
Access to etcd	Access to etcd (the datastore of Kubernetes) should be limited to the control plane only. Depending on your configuration, you should attempt to use etcd over TLS. More information can be found in the etcd documentation .
etcd Encryption	Wherever possible it's a good practice to encrypt all drives at rest, but since etcd holds the state of the entire cluster (including Secrets) its disk should especially be encrypted at rest.

Cluster

There are two areas of concern for securing Kubernetes:

- *Securing the cluster components that are configurable*
- *Securing the applications which run in the cluster*

Components of the Cluster

If you want to protect your cluster from accidental or malicious access and adopt good information practices, read and follow the advice about [securing your cluster](#).

Components in the cluster (your application)

Depending on the attack surface of your application, you may want to focus on specific aspects of security. For example: If you are running a service (Service A) that is critical in a chain of other resources and a separate workload (Service B) which is vulnerable to a resource exhaustion attack then the risk of compromising Service A is high if you do not limit the resources of Service B. The following table lists areas of security concerns and recommendations for securing workloads running in Kubernetes:

Area of Concern for Workload Security	Recommendation
RBAC Authorization (Access to the Kubernetes API)	https://kubernetes.io/docs/reference/access-authn-authz/rbac/
Authentication	https://kubernetes.io/docs/concepts/security/controlling-access/

Area of Concern for Workload Security	Recommendation
Application secrets management (and encrypting them in etcd at rest)	https://kubernetes.io/docs/concepts/configuration/secret/ https://kubernetes.io/docs/tasks/administer-cluster/encrypt-data/
Pod Security Policies	https://kubernetes.io/docs/concepts/policy/pod-security-policy/
Quality of Service (and Cluster resource management)	https://kubernetes.io/docs/tasks/configure-pod-container/quality-service-pod/
Network Policies	https://kubernetes.io/docs/concepts/services-networking/network-policies/
TLS For Kubernetes Ingress	https://kubernetes.io/docs/concepts/services-networking/ingress/#tls

Container

Container security is outside the scope of this guide. Here are general recommendations and links to explore this topic:

Area of Concern for Containers	Recommendation
Container Vulnerability Scanning and OS Dependency Security	As part of an image build step, you should scan your containers for known vulnerabilities.
Image Signing and Enforcement	Sign container images to maintain a system of trust for the content of your containers.
Disallow privileged users	When constructing containers, consult your documentation for how to create users inside of the containers that have the least level of operating system privilege necessary in order to carry out the goal of the container.

Code

Application code is one of the primary attack surfaces over which you have the most control. While securing application code is outside of the Kubernetes security topic, here are recommendations to protect application code:

Code security

Area of Concern for Code	Recommendation
Access over TLS only	If your code needs to communicate by TCP, perform a TLS handshake with the client ahead of time. With the exception of a few cases, encrypt everything in transit. Going one step further, it's a good idea to encrypt network traffic between services. This can be done through a process known as mutual or mTLS which performs a two sided verification of communication between two certificate holding services.
Limiting port ranges of communication	This recommendation may be a bit self-explanatory, but wherever possible you should only expose the ports on your service that are absolutely essential for communication or metric gathering.
3rd Party Dependency Security	It is a good practice to regularly scan your application's third party libraries for known security vulnerabilities. Each programming language has a tool for performing this check automatically.
Static Code Analysis	Most languages provide a way for a snippet of code to be analyzed for any potentially unsafe coding practices. Whenever possible you should perform checks using automated tooling that can scan codebases for common security errors. Some of the tools can be found at: https://owasp.org/www-community/Source_Code_Analysis_Tools
Dynamic probing attacks	There are a few automated tools that you can run against your service to try some of the well known service attacks. These include SQL injection, CSRF, and XSS. One of the most popular dynamic analysis tools is the OWASP Zed Attack proxy tool.

What's next

Learn about related Kubernetes security topics:

- [Pod security standards](#)
- [Network policies for Pods](#)
- [Controlling Access to the Kubernetes API](#)
- [Securing your cluster](#)
- [Data encryption in transit for the control plane](#)
- [Data encryption at rest](#)
- [Secrets in Kubernetes](#)

Feedback

Was this page helpful?

Yes No

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

Last modified October 13, 2020 at 12:41 AM PST: [Transfer "Controlling Access to the Kubernetes API" to the Concepts section \(78351ecaf\)](#)
[Edit this page](#) [Create child page](#) [Create an issue](#)

- [The 4C's of Cloud Native security](#)
- [Cloud](#)
 - [Cloud provider security](#)
 - [Infrastructure security](#)
- [Cluster](#)
 - [Components of the Cluster](#)
 - [Components in the cluster \(your application\)](#)
- [Container](#)
- [Code](#)
 - [Code security](#)
- [What's next](#)

Pod Security Standards

Security settings for Pods are typically applied by using [security contexts](#). Security Contexts allow for the definition of privilege and access controls on a per-Pod basis.

The enforcement and policy-based definition of cluster requirements of security contexts has previously been achieved using [Pod Security Policy](#). A Pod Security Policy is a cluster-level resource that controls security sensitive aspects of the Pod specification.

However, numerous means of policy enforcement have arisen that augment or replace the use of PodSecurityPolicy. The intent of this page is to detail recommended Pod security profiles, decoupled from any specific instantiation.

Policy Types

There is an immediate need for base policy definitions to broadly cover the security spectrum. These should range from highly restricted to highly flexible:

- **Privileged** - Unrestricted policy, providing the widest possible level of permissions. This policy allows for known privilege escalations.
- **Baseline/Default** - Minimally restrictive policy while preventing known privilege escalations. Allows the default (minimally specified) Pod configuration.
- **Restricted** - Heavily restricted policy, following current Pod hardening best practices.

Policies

Privileged

The Privileged policy is purposely-open, and entirely unrestricted. This type of policy is typically aimed at system- and infrastructure-level workloads managed by privileged, trusted users.

The privileged policy is defined by an absence of restrictions. For allow-by-default enforcement mechanisms (such as gatekeeper), the privileged profile may be an absence of applied constraints rather than an instantiated policy. In contrast, for a deny-by-default mechanism (such as Pod Security Policy) the privileged policy should enable all controls (disable all restrictions).

Baseline/Default

The Baseline/Default policy is aimed at ease of adoption for common containerized workloads while preventing known privilege escalations. This policy is targeted at application operators and developers of non-critical applications. The following listed controls should be enforced/disallowed:

Control	Policy
Host Namespaces	Sharing the host namespaces must be disallowed. Restricted Fields: spec.hostNetwork spec.hostPID spec.hostIPC Allowed Values: false
Privileged Containers	Privileged Pods disable most security mechanisms and must be disallowed. Restricted Fields: spec.containers[*].securityContext.privileged spec.initContainers[*].securityContext.privileged Allowed Values: false, undefined/nil
Capabilities	Adding additional capabilities beyond the default set must be disallowed. Restricted Fields: spec.containers[*].securityContext.capabilities.add spec.initContainers[*].securityContext.capabilities.add Allowed Values: empty (or restricted to a known list)

HostPath Volumes	<p>HostPath volumes must be forbidden.</p> <p>Restricted Fields: spec.volumes[*].hostPath</p> <p>Allowed Values: undefined/nil</p>
Host Ports	<p>HostPorts should be disallowed, or at minimum restricted to a known list.</p> <p>Restricted Fields: spec.containers[*].ports[*].hostPort spec.initContainers[*].ports[*].hostPort</p> <p>Allowed Values: 0, undefined (or restricted to a known list)</p>
AppArmor <i>(optional)</i>	<p>On supported hosts, the 'runtime/default' AppArmor profile is applied by default. The default policy should prevent overriding or disabling the policy, or restrict overrides to an allowed set of profiles.</p> <p>Restricted Fields: metadata.annotations['container.apparmor.security.beta.kubernetes.io/apparmor']</p> <p>Allowed Values: 'runtime/default', undefined</p>
SELinux <i>(optional)</i>	<p>Setting custom SELinux options should be disallowed.</p> <p>Restricted Fields: spec.securityContext.seLinuxOptions spec.containers[*].securityContext.seLinuxOptions spec.initContainers[*].securityContext.seLinuxOptions</p> <p>Allowed Values: undefined/nil</p>
/proc Mount Type	<p>The default /proc masks are set up to reduce attack surface, and should be required.</p> <p>Restricted Fields: spec.containers[*].securityContext.procMount spec.initContainers[*].securityContext.procMount</p> <p>Allowed Values: undefined/nil, 'Default'</p>

Sysctls	<p>Sysctls can disable security mechanisms or affect all containers on host, and should be disallowed except for an allowed "safe" subset. sysctl is considered safe if it is namespaced in the container or the Pod, and it is isolated from other Pods or processes on the same Node.</p> <p>Restricted Fields: spec.securityContext.sysctls</p> <p>Allowed Values:</p> <ul style="list-style-type: none"> kernel.shm_rmid_forced net.ipv4.ip_local_port_range net.ipv4.tcp_syncookies net.ipv4.ping_group_range undefined/empty
---------	---

Restricted

The Restricted policy is aimed at enforcing current Pod hardening best practices, at the expense of some compatibility. It is targeted at operators and developers of security-critical applications, as well as lower-trust users. The following listed controls should be enforced/disallowed:

Control	Policy
<i>Everything from the default profile.</i>	

Volume Types	<p>In addition to restricting HostPath volumes, the restricted profile limits usage of non-core volume types to those defined through PersistentVolumes.</p> <p>Restricted Fields:</p> <ul style="list-style-type: none"> spec.volumes[*].hostPath spec.volumes[*].gcePersistentDisk spec.volumes[*].awsElasticBlockStore spec.volumes[*].gitRepo spec.volumes[*].nfs spec.volumes[*].iscsi spec.volumes[*].glusterfs spec.volumes[*].rbd spec.volumes[*].flexVolume spec.volumes[*].cinder spec.volumes[*].cephFS spec.volumes[*].flocker spec.volumes[*].fc spec.volumes[*].azureFile spec.volumes[*].vsphereVolume spec.volumes[*].quobyte spec.volumes[*].azureDisk spec.volumes[*].portworxVolume spec.volumes[*].scaleIO spec.volumes[*].storageos spec.volumes[*].csi <p>Allowed Values: undefined/nil</p>
Privilege Escalation	<p>Privilege escalation (such as via set-user-ID or set-group-ID file mode) should not be allowed.</p> <p>Restricted Fields:</p> <ul style="list-style-type: none"> spec.containers[*].securityContext.allowPrivilegeEscalation spec.initContainers[*].securityContext.allowPrivilegeEscalation <p>Allowed Values: false</p>
Running as Non-root	<p>Containers must be required to run as non-root users.</p> <p>Restricted Fields:</p> <ul style="list-style-type: none"> spec.securityContext.runAsNonRoot spec.containers[*].securityContext.runAsNonRoot spec.initContainers[*].securityContext.runAsNonRoot <p>Allowed Values: true</p>

Non-root groups (optional)	<p>Containers should be forbidden from running with a root primary or supplementary GID.</p> <p>Restricted Fields:</p> <ul style="list-style-type: none"> spec.securityContext.runAsGroup spec.securityContext.supplementalGroups[*] spec.securityContext.fsGroup spec.containers[*].securityContext.runAsGroup spec.initContainers[*].securityContext.runAsGroup <p>Allowed Values:</p> <ul style="list-style-type: none"> non-zero undefined / nil (except for `*.runAsGroup`)
Seccomp	<p>The RuntimeDefault seccomp profile must be required, or allow specific additional profiles.</p> <p>Restricted Fields:</p> <ul style="list-style-type: none"> spec.securityContext.seccompProfile.type spec.containers[*].securityContext.seccompProfile spec.initContainers[*].securityContext.seccompProfile <p>Allowed Values:</p> <ul style="list-style-type: none"> 'runtime/default' undefined / nil

Policy Instantiation

Decoupling policy definition from policy instantiation allows for a common understanding and consistent language of policies across clusters, independent of the underlying enforcement mechanism.

As mechanisms mature, they will be defined below on a per-policy basis. The methods of enforcement of individual policies are not defined here.

PodSecurityPolicy

- [Privileged](#)
- [Baseline](#)
- [Restricted](#)

FAQ

Why isn't there a profile between privileged and default?

The three profiles defined here have a clear linear progression from most secure (restricted) to least secure (privileged), and cover a broad set of workloads. Privileges required above the baseline policy are typically very application specific, so we do not offer a standard profile in this niche. This is not to say that the privileged profile should always be used in this case, but that policies in this space need to be defined on a case-by-case basis.

SIG Auth may reconsider this position in the future, should a clear need for other profiles arise.

What's the difference between a security policy and a security context?

[Security Contexts](#) configure Pods and Containers at runtime. Security contexts are defined as part of the Pod and container specifications in the Pod manifest, and represent parameters to the container runtime.

Security policies are control plane mechanisms to enforce specific settings in the Security Context, as well as other parameters outside the Security Context. As of February 2020, the current native solution for enforcing these security policies is [Pod Security Policy](#) - a mechanism for centrally enforcing security policy on Pods across a cluster. Other alternatives for enforcing security policy are being developed in the Kubernetes ecosystem, such as [OPA Gatekeeper](#).

What profiles should I apply to my Windows Pods?

Windows in Kubernetes has some limitations and differentiators from standard Linux-based workloads. Specifically, the Pod SecurityContext fields [have no effect on Windows](#). As such, no standardized Pod Security profiles currently exists.

What about sandboxed Pods?

There is not currently an API standard that controls whether a Pod is considered sandboxed or not. Sandbox Pods may be identified by the use of a sandboxed runtime (such as gVisor or Kata Containers), but there is no standard definition of what a sandboxed runtime is.

The protections necessary for sandboxed workloads can differ from others. For example, the need to restrict privileged permissions is lessened when the workload is isolated from the underlying kernel. This allows for workloads requiring heightened permissions to still be isolated.

Additionally, the protection of sandboxed workloads is highly dependent on the method of sandboxing. As such, no single recommended policy is recommended for all sandboxed workloads.

Feedback

Was this page helpful?

Yes No

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

Last modified September 19, 2020 at 4:48 PM PST: [Context to Context \(70eba58d3\)](#)

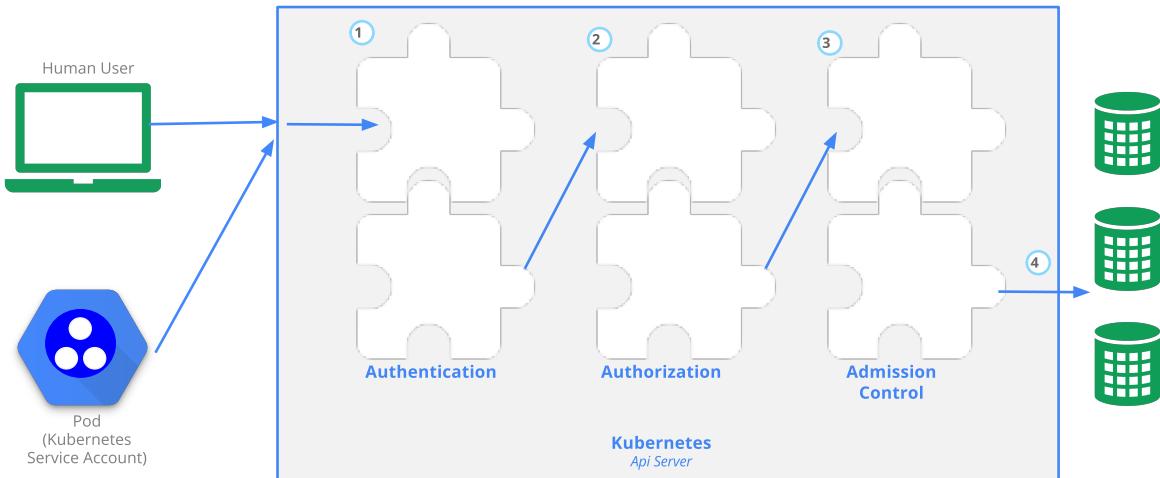
[Edit this page](#) [Create child page](#) [Create an issue](#)

- [Policy Types](#)
- [Policies](#)
 - [Privileged](#)
 - [Baseline/Default](#)
 - [Restricted](#)
- [Policy Instantiation](#)
- [FAQ](#)
 - [Why isn't there a profile between privileged and default?](#)
 - [What's the difference between a security policy and a security context?](#)
 - [What profiles should I apply to my Windows Pods?](#)
 - [What about sandboxed Pods?](#)

Controlling Access to the Kubernetes API

This page provides an overview of controlling access to the Kubernetes API.

Users access the [Kubernetes API](#) using `kubectl`, client libraries, or by making REST requests. Both human users and [Kubernetes service accounts](#) can be authorized for API access. When a request reaches the API, it goes through several stages, illustrated in the following diagram:



Transport security

In a typical Kubernetes cluster, the API serves on port 443, protected by TLS. The API server presents a certificate. This certificate may be signed using a private certificate authority (CA), or based on a public key infrastructure linked to a generally recognized CA.

If your cluster uses a private certificate authority, you need a copy of that CA certificate configured into your `~/.kube/config` on the client, so that you can trust the connection and be confident it was not intercepted.

Your client can present a TLS client certificate at this stage.

Authentication

Once TLS is established, the HTTP request moves to the Authentication step. This is shown as step 1 in the diagram. The cluster creation script or cluster admin configures the API server to run one or more Authenticator modules. Authenticators are described in more detail in [Authentication](#).

The input to the authentication step is the entire HTTP request; however, it typically just examines the headers and/or client certificate.

Authentication modules include client certificates, password, and plain tokens, bootstrap tokens, and JSON Web Tokens (used for service accounts).

Multiple authentication modules can be specified, in which case each one is tried in sequence, until one of them succeeds.

If the request cannot be authenticated, it is rejected with HTTP status code 401. Otherwise, the user is authenticated as a specific `username`, and the user name is available to subsequent steps to use in their decisions. Some authenticators also provide the group memberships of the user, while other authenticators do not.

While Kubernetes uses usernames for access control decisions and in request logging, it does not have a `User` object nor does it store usernames or other information about users in its API.

Authorization

After the request is authenticated as coming from a specific user, the request must be authorized. This is shown as step 2 in the diagram.

A request must include the username of the requester, the requested action, and the object affected by the action. The request is authorized if an existing policy declares that the user has permissions to complete the requested action.

For example, if Bob has the policy below, then he can read pods only in the namespace `projectCaribou`:

```
{  
    "apiVersion": "abac.authorization.kubernetes.io/v1beta1",  
    "kind": "Policy",  
    "spec": {  
        "user": "bob",  
        "namespace": "projectCaribou",  
        "resource": "pods",  
        "readonly": true  
    }  
}
```

If Bob makes the following request, the request is authorized because he is allowed to read objects in the projectCaribou namespace:

```
{  
    "apiVersion": "authorization.k8s.io/v1beta1",  
    "kind": "SubjectAccessReview",  
    "spec": {  
        "resourceAttributes": {  
            "namespace": "projectCaribou",  
            "verb": "get",  
            "group": "unicorn.example.org",  
            "resource": "pods"  
        }  
    }  
}
```

If Bob makes a request to write (create or update) to the objects in the projectCaribou namespace, his authorization is denied. If Bob makes a request to read (get) objects in a different namespace such as projectFish, then his authorization is denied.

Kubernetes authorization requires that you use common REST attributes to interact with existing organization-wide or cloud-provider-wide access control systems. It is important to use REST formatting because these control systems might interact with other APIs besides the Kubernetes API.

Kubernetes supports multiple authorization modules, such as ABAC mode, RBAC Mode, and Webhook mode. When an administrator creates a cluster, they configure the authorization modules that should be used in the API server. If more than one authorization modules are configured, Kubernetes checks each module, and if any module authorizes the request, then the request can proceed. If all of the modules deny the request, then the request is denied (HTTP status code 403).

To learn more about Kubernetes authorization, including details about creating policies using the supported authorization modules, see [Authorization](#).

Admission control

Admission Control modules are software modules that can modify or reject requests. In addition to the attributes available to Authorization modules, Admission Control modules can access the contents of the object that is being created or modified.

Admission controllers act on requests that create, modify, delete, or connect to (proxy) an object. Admission controllers do not act on requests that merely read objects. When multiple admission controllers are configured, they are called in order.

This is shown as step 3 in the diagram.

Unlike Authentication and Authorization modules, if any admission controller module rejects, then the request is immediately rejected.

In addition to rejecting objects, admission controllers can also set complex defaults for fields.

The available Admission Control modules are described in [Admission Controllers](#).

Once a request passes all admission controllers, it is validated using the validation routines for the corresponding API object, and then written to the object store (shown as step 4).

API server ports and IPs

The previous discussion applies to requests sent to the secure port of the API server (the typical case). The API server can actually serve on 2 ports:

By default the Kubernetes API server serves HTTP on 2 ports:

1. *localhost port:*

- *is intended for testing and bootstrap, and for other components of the master node (scheduler, controller-manager) to talk to the API*
- *no TLS*
- *default is port 8080, change with --insecure-port flag.*
- *default IP is localhost, change with --insecure-bind-address flag.*
- *request **bypasses** authentication and authorization modules.*
- *request handled by admission control module(s).*
- *protected by need to have host access*

2. *"Secure port":*

- *use whenever possible*
- *uses TLS. Set cert with --tls-cert-file and key with --tls-private-key-file flag.*
- *default is port 6443, change with --secure-port flag.*

- default IP is first non-localhost network interface, change with `--bind-address` flag.
- request handled by authentication and authorization modules.
- request handled by admission control module(s).
- authentication and authorization modules run.

What's next

Read more documentation on authentication, authorization and API access control:

- [Authenticating](#)
 - [Authenticating with Bootstrap Tokens](#)
- [Admission Controllers](#)
 - [Dynamic Admission Control](#)
- [Authorization](#)
 - [Role Based Access Control](#)
 - [Attribute Based Access Control](#)
 - [Node Authorization](#)
 - [Webhook Authorization](#)
- [Certificate Signing Requests](#)
 - including [CSR approval](#) and [certificate signing](#)
- Service accounts
 - [Developer guide](#)
 - [Administration](#)

You can learn about:

- how Pods can use [Secrets](#) to obtain API credentials.

Feedback

Was this page helpful?

Yes No

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

Last modified October 13, 2020 at 12:41 AM PST: [Transfer "Controlling Access to the Kubernetes API" to the Concepts section \(78351ecaf\)](#)
[Edit this page](#) [Create child page](#) [Create an issue](#)

- [Transport security](#)
- [Authentication](#)
- [Authorization](#)
- [Admission control](#)
- [API server ports and IPs](#)
- [What's next](#)

Policies

Policies you can configure that apply to groups of resources.

[**Limit Ranges**](#)

[**Resource Quotas**](#)

[**Pod Security Policies**](#)

[**Process ID Limits And Reservations**](#)

Limit Ranges

By default, containers run with unbounded [compute resources](#) on a Kubernetes cluster. With resource quotas, cluster administrators can restrict resource consumption and creation on a [namespace](#) basis. Within a namespace, a Pod or Container can consume as much CPU and memory as defined by the namespace's resource quota. There is a concern that one Pod or Container could monopolize all available resources. A LimitRange is a policy to constrain resource allocations (to Pods or Containers) in a namespace.

A LimitRange provides constraints that can:

- Enforce minimum and maximum compute resources usage per Pod or Container in a namespace.
- Enforce minimum and maximum storage request per PersistentVolumeClaim in a namespace.
- Enforce a ratio between request and limit for a resource in a namespace.
- Set default request/limit for compute resources in a namespace and automatically inject them to Containers at runtime.

Enabling LimitRange

LimitRange support has been enabled by default since Kubernetes 1.10.

A LimitRange is enforced in a particular namespace when there is a LimitRange object in that namespace.

The name of a LimitRange object must be a valid [DNS subdomain name](#).

Overview of Limit Range

- The administrator creates one LimitRange in one namespace.
- Users create resources like Pods, Containers, and PersistentVolumeClaims in the namespace.

- The *LimitRanger* admission controller enforces defaults and limits for all Pods and Containers that do not set compute resource requirements and tracks usage to ensure it does not exceed resource minimum, maximum and ratio defined in any *LimitRange* present in the namespace.
- If creating or updating a resource (Pod, Container, PersistentVolumeClaim) that violates a *LimitRange* constraint, the request to the API server will fail with an HTTP status code 403 FORBIDDEN and a message explaining the constraint that have been violated.
- If a *LimitRange* is activated in a namespace for compute resources like *cpu* and *memory*, users must specify requests or limits for those values. Otherwise, the system may reject Pod creation.
- *LimitRange* validations occurs only at Pod Admission stage, not on Running Pods.

Examples of policies that could be created using limit range are:

- In a 2 node cluster with a capacity of 8 GiB RAM and 16 cores, constrain Pods in a namespace to request 100m of CPU with a max limit of 500m for CPU and request 200Mi for Memory with a max limit of 600Mi for Memory.
- Define default CPU limit and request to 150m and memory default request to 300Mi for Containers started with no *cpu* and *memory* requests in their specs.

In the case where the total limits of the namespace is less than the sum of the limits of the Pods/Containers, there may be contention for resources. In this case, the Containers or Pods will not be created.

Neither contention nor changes to a *LimitRange* will affect already created resources.

What's next

Refer to the [LimitRanger design document](#) for more information.

For examples on using limits, see:

- [how to configure minimum and maximum CPU constraints per namespace](#).
- [how to configure minimum and maximum Memory constraints per namespace](#).
- [how to configure default CPU Requests and Limits per namespace](#).
- [how to configure default Memory Requests and Limits per namespace](#).
- [how to configure minimum and maximum Storage consumption per namespace](#).
- a [detailed example on configuring quota per namespace](#).

Feedback

Was this page helpful?

Yes No

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

Last modified July 17, 2020 at 4:11 PM PST: [Fix links in concepts section \(2\)](#) ([c8f470487](#))

[Edit this page](#) [Create child page](#) [Create an issue](#)

- [Enabling LimitRange](#)
 - [Overview of Limit Range](#)
- [What's next](#)

Resource Quotas

When several users or teams share a cluster with a fixed number of nodes, there is a concern that one team could use more than its fair share of resources.

Resource quotas are a tool for administrators to address this concern.

A resource quota, defined by a `ResourceQuota` object, provides constraints that limit aggregate resource consumption per namespace. It can limit the quantity of objects that can be created in a namespace by type, as well as the total amount of compute resources that may be consumed by resources in that namespace.

Resource quotas work like this:

- Different teams work in different namespaces. Currently this is voluntary, but support for making this mandatory via ACLs is planned.
- The administrator creates one `ResourceQuota` for each namespace.
- Users create resources (pods, services, etc.) in the namespace, and the quota system tracks usage to ensure it does not exceed hard resource limits defined in a `ResourceQuota`.
- If creating or updating a resource violates a quota constraint, the request will fail with HTTP status code `403 FORBIDDEN` with a message explaining the constraint that would have been violated.
- If quota is enabled in a namespace for compute resources like `cpu` and `memory`, users must specify requests or limits for those values; otherwise, the quota system may reject pod creation. Hint: Use the `LimitRanger` admission controller to force defaults for pods that make no compute resource requirements.

See the [walkthrough](#) for an example of how to avoid this problem.

The name of a ResourceQuota object must be a valid [DNS subdomain name](#).

Examples of policies that could be created using namespaces and quotas are:

- In a cluster with a capacity of 32 GiB RAM, and 16 cores, let team A use 20 GiB and 10 cores, let B use 10GiB and 4 cores, and hold 2GiB and 2 cores in reserve for future allocation.
- Limit the "testing" namespace to using 1 core and 1GiB RAM. Let the "production" namespace use any amount.

In the case where the total capacity of the cluster is less than the sum of the quotas of the namespaces, there may be contention for resources. This is handled on a first-come-first-served basis.

Neither contention nor changes to quota will affect already created resources.

Enabling Resource Quota

Resource Quota support is enabled by default for many Kubernetes distributions. It is enabled when the API server --enable-admission-plugins= flag has ResourceQuota as one of its arguments.

A resource quota is enforced in a particular namespace when there is a ResourceQuota in that namespace.

Compute Resource Quota

You can limit the total sum of [compute resources](#) that can be requested in a given namespace.

The following resource types are supported:

Resource Name	Description
limits.cpu	Across all pods in a non-terminal state, the sum of CPU limits cannot exceed this value.
limits.memory	Across all pods in a non-terminal state, the sum of memory limits cannot exceed this value.
requests.cpu	Across all pods in a non-terminal state, the sum of CPU requests cannot exceed this value.
requests.memory	Across all pods in a non-terminal state, the sum of memory requests cannot exceed this value.
hugepages-<size>	Across all pods in a non-terminal state, the number of huge page requests of the specified size cannot exceed this value.
cpu	Same as requests.cpu
memory	Same as requests.memory

Resource Quota For Extended Resources

In addition to the resources mentioned above, in release 1.10, quota support for [extended resources](#) is added.

As overcommit is not allowed for extended resources, it makes no sense to specify both requests and limits for the same extended resource in a quota. So for extended resources, only quota items with prefix requests. is allowed for now.

Take the GPU resource as an example, if the resource name is nvidia.com/gpu, and you want to limit the total number of GPUs requested in a namespace to 4, you can define a quota as follows:

- `requests.nvidia.com/gpu: 4`

See [Viewing and Setting Quotas](#) for more detail information.

Storage Resource Quota

You can limit the total sum of [storage resources](#) that can be requested in a given namespace.

In addition, you can limit consumption of storage resources based on associated storage-class.

Resource Name	Description
<code>requests.storage</code>	Across all persistent volume claims, the sum of storage requests cannot exceed this value.
<code>persistentvolumeclaims</code>	The total number of PersistentVolumeClaims that can exist in the namespace.
<code><storage-class-name>.storageclass.storage.k8s.io/requests.storage</code>	Across all persistent volume claims associated with the <storage-class-name>, the sum of storage requests cannot exceed this value.
<code><storage-class-name>.storageclass.storage.k8s.io/persistentvolumeclaims</code>	Across all persistent volume claims associated with the storage-class-name, the total number of persistent volume claims that can exist in the namespace.

For example, if an operator wants to quota storage with gold storage class separate from bronze storage class, the operator can define a quota as follows:

- `gold.storageclass.storage.k8s.io/requests.storage: 500Gi`

- `bronze.storageclass.storage.k8s.io/requests.storage: 100Gi`

In release 1.8, quota support for local ephemeral storage is added as an alpha feature:

Resource Name	Description
<code>requests.ephemeral-storage</code>	Across all pods in the namespace, the sum of local ephemeral storage requests cannot exceed this value.
<code>limits.ephemeral-storage</code>	Across all pods in the namespace, the sum of local ephemeral storage limits cannot exceed this value.
<code>ephemeral-storage</code>	Same as <code>requests.ephemeral-storage</code> .

Object Count Quota

You can set quota for the total number of certain resources of all standard, namespaced resource types using the following syntax:

- `count/<resource>.<group>` for resources from non-core groups
- `count/<resource>` for resources from the core group

Here is an example set of resources users may want to put under object count quota:

- `count/persistentvolumeclaims`
- `count/services`
- `count/secrets`
- `count/configmaps`
- `count/replicationcontrollers`
- `count/deployments.apps`
- `count/replicasets.apps`
- `count/statefulsets.apps`
- `count/jobs.batch`
- `count/cronjobs.batch`

The same syntax can be used for custom resources. For example, to create a quota on a `widgets` custom resource in the `example.com` API group, use `count/widgets.example.com`.

When using `count/` resource quota, an object is charged against the quota if it exists in server storage. These types of quotas are useful to protect against exhaustion of storage resources. For example, you may want to limit the number of Secrets in a server given their large size. Too many Secrets in a cluster can actually prevent servers and controllers from starting. You can set a quota for Jobs to protect against a poorly configured CronJob. CronJobs that create too many Jobs in a namespace can lead to a denial of service.*

It is also possible to do generic object count quota on a limited set of resources. The following types are supported:

Resource Name	Description
configmaps	The total number of ConfigMaps that can exist in the namespace.
persistentvolumeclaims	The total number of PersistentVolumeClaims that can exist in the namespace.
pods	The total number of Pods in a non-terminal state that can exist in the namespace. A pod is in a terminal state if <code>.status.phase</code> in (<code>Failed</code> , <code>Succeeded</code>) is true.
replicationcontrollers	The total number of ReplicationControllers that can exist in the namespace.
resourcequotas	The total number of ResourceQuotas that can exist in the namespace.
services	The total number of Services that can exist in the namespace.
services.loadbalancers	The total number of Services of type <code>LoadBalancer</code> that can exist in the namespace.
services.nodeports	The total number of Services of type <code>NodePort</code> that can exist in the namespace.
secrets	The total number of Secrets that can exist in the namespace.

For example, pods quota counts and enforces a maximum on the number of pods created in a single namespace that are not terminal. You might want to set a pods quota on a namespace to avoid the case where a user creates many small pods and exhausts the cluster's supply of Pod IPs.

Quota Scopes

Each quota can have an associated set of scopes. A quota will only measure usage for a resource if it matches the intersection of enumerated scopes.

When a scope is added to the quota, it limits the number of resources it supports to those that pertain to the scope. Resources specified on the quota outside of the allowed set results in a validation error.

Scope	Description
Terminating	Match pods where <code>.spec.activeDeadlineSeconds >= 0</code>
NotTerminating	Match pods where <code>.spec.activeDeadlineSeconds</code> is nil
BestEffort	Match pods that have best effort quality of service.
NotBestEffort	Match pods that do not have best effort quality of service.
PriorityClass	Match pods that references the specified priority class .

The BestEffort scope restricts a quota to tracking the following resource:

- pods

The `Terminating`, `NotTerminating`, `NotBestEffort` and `PriorityClass` scopes restrict a quota to tracking the following resources:

- `pods`
- `cpu`
- `memory`
- `requests.cpu`
- `requests.memory`
- `limits.cpu`
- `limits.memory`

Note that you cannot specify both the `Terminating` and the `NotTerminating` scopes in the same quota, and you cannot specify both the `BestEffort` and `NotBestEffort` scopes in the same quota either.

The `scopeSelector` supports the following values in the `operator` field:

- `In`
- `NotIn`
- `Exists`
- `DoesNotExist`

When using one of the following values as the `scopeName` when defining the `scopeSelector`, the `operator` must be `Exists`.

- `Terminating`
- `NotTerminating`
- `BestEffort`
- `NotBestEffort`

If the `operator` is `In` or `NotIn`, the `values` field must have at least one value. For example:

```
scopeSelector:  
  matchExpressions:  
    - scopeName: PriorityClass  
      operator: In  
      values:  
        - middle
```

If the `operator` is `Exists` or `DoesNotExist`, the `values` field must NOT be specified.

Resource Quota Per PriorityClass

FEATURE STATE: Kubernetes v1.17 [stable]

Pods can be created at a specific [priority](#). You can control a pod's consumption of system resources based on a pod's priority, by using the `scopeSelector` field in the quota spec.

A quota is matched and consumed only if `scopeSelector` in the quota spec selects the pod.

When quota is scoped for priority class using scopeSelector field, quota object is restricted to track only following resources:

- *pods*
- *cpu*
- *memory*
- *ephemeral-storage*
- *limits.cpu*
- *limits.memory*
- *limits.ephemeral-storage*
- *requests.cpu*
- *requests.memory*
- *requests.ephemeral-storage*

This example creates a quota object and matches it with pods at specific priorities. The example works as follows:

- *Pods in the cluster have one of the three priority classes, "low", "medium", "high".*
- *One quota object is created for each priority.*

Save the following YAML to a file quota.yml.

```
apiVersion: v1
kind: List
items:
- apiVersion: v1
  kind: ResourceQuota
  metadata:
    name: pods-high
  spec:
    hard:
      cpu: "1000"
      memory: 200Gi
      pods: "10"
    scopeSelector:
      matchExpressions:
        - operator : In
          scopeName: PriorityClass
          values: ["high"]
- apiVersion: v1
  kind: ResourceQuota
  metadata:
    name: pods-medium
  spec:
    hard:
      cpu: "10"
      memory: 20Gi
      pods: "10"
    scopeSelector:
      matchExpressions:
        - operator : In
```

```

    scopeName: PriorityClass
    values: [ "medium" ]
- apiVersion: v1
  kind: ResourceQuota
  metadata:
    name: pods-low
  spec:
    hard:
      cpu: "5"
      memory: 10Gi
      pods: "10"
    scopeSelector:
      matchExpressions:
        - operator : In
        scopeName: PriorityClass
        values: [ "low" ]

```

Apply the YAML using `kubectl create`.

```
kubectl create -f ./quota.yml
```

```
resourcequota/pods-high created
resourcequota/pods-medium created
resourcequota/pods-low created
```

Verify that *Used* quota is 0 using `kubectl describe quota`.

```
kubectl describe quota
```

Name:	pods-high	
Namespace:	default	
Resource	Used	Hard
cpu	0	1k
memory	0	200Gi
pods	0	10

Name:	pods-low	
Namespace:	default	
Resource	Used	Hard
cpu	0	5
memory	0	10Gi
pods	0	10

Name:	pods-medium	
Namespace:	default	
Resource	Used	Hard
cpu	0	10

```
memory      0    20Gi
pods        0    10
```

Create a pod with priority "high". Save the following YAML to a file `high-priority-pod.yml`.

```
apiVersion: v1
kind: Pod
metadata:
  name: high-priority
spec:
  containers:
  - name: high-priority
    image: ubuntu
    command: ["/bin/sh"]
    args: ["-c", "while true; do echo hello; sleep 10;done"]
    resources:
      requests:
        memory: "10Gi"
        cpu: "500m"
      limits:
        memory: "10Gi"
        cpu: "500m"
  priorityClassName: high
```

Apply it with `kubectl create`.

```
kubectl create -f ./high-priority-pod.yml
```

Verify that "Used" stats for "high" priority quota, `pods-high`, has changed and that the other two quotas are unchanged.

```
kubectl describe quota
```

```
Name:      pods-high
Namespace: default
Resource   Used   Hard
-----  -----
cpu        500m   1k
memory     10Gi   200Gi
pods       1       10
```

```
Name:      pods-low
Namespace: default
Resource   Used   Hard
-----  -----
cpu        0       5
memory     0       10Gi
pods       0       10
```

```
Name:      pods-medium
```

Namespace:	default	
Resource	Used	Hard
cpu	0	10
memory	0	20Gi
pods	0	10

Requests compared to Limits

When allocating compute resources, each container may specify a request and a limit value for either CPU or memory. The quota can be configured to quota either value.

If the quota has a value specified for `requests.cpu` or `requests.memory`, then it requires that every incoming container makes an explicit request for those resources. If the quota has a value specified for `limits.cpu` or `limits.memory`, then it requires that every incoming container specifies an explicit limit for those resources.

Viewing and Setting Quotas

Kubectl supports creating, updating, and viewing quotas:

```
kubectl create namespace myspace
```

```
cat <<EOF > compute-resources.yaml
apiVersion: v1
kind: ResourceQuota
metadata:
  name: compute-resources
spec:
  hard:
    requests.cpu: "1"
    requests.memory: 1Gi
    limits.cpu: "2"
    limits.memory: 2Gi
    requests.nvidia.com/gpu: 4
EOF
```

```
kubectl create -f ./compute-resources.yaml --namespace=myspace
```

```
cat <<EOF > object-counts.yaml
apiVersion: v1
kind: ResourceQuota
metadata:
  name: object-counts
spec:
  hard:
    configmaps: "10"
    persistentvolumeclaims: "4"
    pods: "4"
```

```
replicationcontrollers: "20"
secrets: "10"
services: "10"
services.loadbalancers: "2"
EOF
```

```
kubectl create -f ./object-counts.yaml --namespace=myspace
```

```
kubectl get quota --namespace=myspace
```

NAME	AGE
compute-resources	30s
object-counts	32s

```
kubectl describe quota compute-resources --namespace=myspace
```

Name:	compute-resources	
Namespace:	myspace	
Resource	Used	Hard
-----	-----	-----
limits.cpu	0	2
limits.memory	0	2Gi
requests.cpu	0	1
requests.memory	0	1Gi
requests.nvidia.com/gpu	0	4

```
kubectl describe quota object-counts --namespace=myspace
```

Name:	object-counts	
Namespace:	myspace	
Resource	Used	Hard
-----	-----	-----
configmaps	0	10
persistentvolumeclaims	0	4
pods	0	4
replicationcontrollers	0	20
secrets	1	10
services	0	10
services.loadbalancers	0	2

Kubectl also supports object count quota for all standard namespaced resources using the syntax `count/<resource>. <group>`:

```
kubectl create namespace myspace
```

```
kubectl create quota test --hard=count/deployments.apps=2, count/
replicasets.apps=4, count/pods=3, count/secrets=4 --namespace=myspace
```

```
kubectl create deployment nginx --image=nginx --
namespace=myspace --replicas=2
```

```
kubectl describe quota --namespace=myspace
```

Name:	test	
Namespace:	myspace	
Resource	Used	Hard
-----	-----	-----
count/deployments.apps	1	2
count/pods	2	3
count/replicasets.apps	1	4
count/secrets	1	4

Quota and Cluster Capacity

ResourceQuotas are independent of the cluster capacity. They are expressed in absolute units. So, if you add nodes to your cluster, this does not automatically give each namespace the ability to consume more resources.

Sometimes more complex policies may be desired, such as:

- Proportionally divide total cluster resources among several teams.
- Allow each tenant to grow resource usage as needed, but have a generous limit to prevent accidental resource exhaustion.
- Detect demand from one namespace, add nodes, and increase quota.

Such policies could be implemented using *ResourceQuotas* as building blocks, by writing a "controller" that watches the quota usage and adjusts the quota hard limits of each namespace according to other signals.

Note that resource quota divides up aggregate cluster resources, but it creates no restrictions around nodes: pods from several namespaces may run on the same node.

Limit Priority Class consumption by default

It may be desired that pods at a particular priority, eg. "cluster-services", should be allowed in a namespace, if and only if, a matching quota object exists.

With this mechanism, operators are able to restrict usage of certain high priority classes to a limited number of namespaces and not every namespace will be able to consume these priority classes by default.

To enforce this, `kube-apiserver` flag `--admission-control-config-file` should be used to pass path to the following configuration file:

```
apiVersion: apiserver.config.k8s.io/v1
kind: AdmissionConfiguration
plugins:
- name: "ResourceQuota"
  configuration:
    apiVersion: apiserver.config.k8s.io/v1
    kind: ResourceQuotaConfiguration
    limitedResources:
```

```
- resource: pods
  matchScopes:
    - scopeName: PriorityClass
      operator: In
      values: ["cluster-services"]
```

Now, "cluster-services" pods will be allowed in only those namespaces where a quota object with a matching scopeSelector is present. For example:

```
scopeSelector:
  matchExpressions:
    - scopeName: PriorityClass
      operator: In
      values: ["cluster-services"]
```

What's next

- See [ResourceQuota design doc](#) for more information.
- See a [detailed example for how to use resource quota](#).
- Read [Quota support for priority class design doc](#).
- See [LimitedResources](#)

Feedback

Was this page helpful?

Yes No

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

Last modified November 05, 2020 at 12:12 PM PST: [\[zh\] Sync changes to docs/concepts/policy/resource-quotas.md \(4bf6c16cc\)](#)

[Edit this page](#) [Create child page](#) [Create an issue](#)

- [Enabling Resource Quota](#)
- [Compute Resource Quota](#)
 - [Resource Quota For Extended Resources](#)
- [Storage Resource Quota](#)
- [Object Count Quota](#)
- [Quota Scopes](#)
 - [Resource Quota Per PriorityClass](#)
- [Requests compared to Limits](#)
- [Viewing and Setting Quotas](#)
- [Quota and Cluster Capacity](#)
- [Limit Priority Class consumption by default](#)
- [What's next](#)

Pod Security Policies

FEATURE STATE: Kubernetes v1.20 [beta]

Pod Security Policies enable fine-grained authorization of pod creation and updates.

What is a Pod Security Policy?

A Pod Security Policy is a cluster-level resource that controls security sensitive aspects of the pod specification. The [PodSecurityPolicy](#) objects define a set of conditions that a pod must run with in order to be accepted into the system, as well as defaults for the related fields. They allow an administrator to control the following:

Control Aspect	Field Names
Running of privileged containers	privileged
Usage of host namespaces	hostPID , hostIPC
Usage of host networking and ports	hostNetwork , hostPorts
Usage of volume types	volumes
Usage of the host filesystem	allowedHostPaths
Allow specific FlexVolume drivers	allowedFlexVolumes
Allocating an FSGroup that owns the pod's volumes	fsGroup
Requiring the use of a read only root file system	readOnlyRootFilesystem

Control Aspect	Field Names
The user and group IDs of the container	runAsUser , runAsGroup , supplementalGroups
Restricting escalation to root privileges	allowPrivilegeEscalation , defaultAllowPrivilegeEscalation
Linux capabilities	defaultAddCapabilities , requiredDropCapabilities , allowedCapabilities
The SELinux context of the container	seLinux
The Allowed Proc Mount types for the container	allowedProcMountTypes
The AppArmor profile used by containers	annotations
The seccomp profile used by containers	annotations
The sysctl profile used by containers	forbiddenSysctls , allowedUnsafeSysctls

Enabling Pod Security Policies

*Pod security policy control is implemented as an optional (but recommended) [admission controller](#). PodSecurityPolicies are enforced by [enabling the admission controller](#), but doing so without authorizing any policies **will prevent any pods from being created** in the cluster.*

Since the pod security policy API (`policy/v1beta1/podsecuritypolicy`) is enabled independently of the admission controller, for existing clusters it is recommended that policies are added and authorized before enabling the admission controller.

Authorizing Policies

When a `PodSecurityPolicy` resource is created, it does nothing. In order to use it, the requesting user or target pod's [service account](#) must be authorized to use the policy, by allowing the `use` verb on the policy.

Most Kubernetes pods are not created directly by users. Instead, they are typically created indirectly as part of a [Deployment](#), [ReplicaSet](#), or other templated controller via the controller manager. Granting the controller access to the policy would grant access for all pods created by that controller, so the preferred method for authorizing policies is to grant access to the pod's service account (see [example](#)).

Via RBAC

[RBAC](#) is a standard Kubernetes authorization mode, and can easily be used to authorize use of policies.

First, a `Role` or `ClusterRole` needs to grant access to use the desired policies. The rules to grant access look like this:

```
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRole
metadata:
  name: <role name>
rules:
- apiGroups: ['policy']
  resources: ['podsecuritypolicies']
  verbs:     ['use']
  resourceNames:
  - <list of policies to authorize>
```

Then the (`Cluster`)`Role` is bound to the authorized user(s):

```
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRoleBinding
metadata:
  name: <binding name>
roleRef:
  kind: ClusterRole
  name: <role name>
  apiGroup: rbac.authorization.k8s.io
subjects:
# Authorize specific service accounts:
- kind: ServiceAccount
  name: <authorized service account name>
  namespace: <authorized pod namespace>
# Authorize specific users (not recommended):
- kind: User
  apiGroup: rbac.authorization.k8s.io
  name: <authorized user name>
```

If a `RoleBinding` (not a `ClusterRoleBinding`) is used, it will only grant usage for pods being run in the same namespace as the binding. This can be paired with system groups to grant access to all pods run in the namespace:

```
# Authorize all service accounts in a namespace:  
- kind: Group  
  apiGroup: rbac.authorization.k8s.io  
  name: system:serviceaccounts  
# Or equivalently, all authenticated users in a namespace:  
- kind: Group  
  apiGroup: rbac.authorization.k8s.io  
  name: system:authenticated
```

For more examples of RBAC bindings, see [Role Binding Examples](#). For a complete example of authorizing a `PodSecurityPolicy`, see [below](#).

Troubleshooting

- The [controller manager](#) must be run against the secured API port and must not have superuser permissions. See [Controlling Access to the Kubernetes API](#) to learn about API server access controls.
If the controller manager connected through the trusted API port (also known as the `localhost` listener), requests would bypass authentication and authorization modules; all `PodSecurityPolicy` objects would be allowed, and users would be able to create grant themselves the ability to create privileged containers.

For more details on configuring controller manager authorization, see [Controller Roles](#).

Policy Order

In addition to restricting pod creation and update, pod security policies can also be used to provide default values for many of the fields that it controls. When multiple policies are available, the pod security policy controller selects policies according to the following criteria:

1. *PodSecurityPolicies which allow the pod as-is, without changing defaults or mutating the pod, are preferred. The order of these non-mutating PodSecurityPolicies doesn't matter.*
2. *If the pod must be defaulted or mutated, the first PodSecurityPolicy (ordered by name) to allow the pod is selected.*

Note: During update operations (during which mutations to pod specs are disallowed) only non-mutating PodSecurityPolicies are used to validate the pod.

Example

This example assumes you have a running cluster with the `PodSecurityPolicy` admission controller enabled and you have cluster admin privileges.

Set up

Set up a namespace and a service account to act as for this example. We'll use this service account to mock a non-admin user.

```
kubectl create namespace psp-example  
kubectl create serviceaccount -n psp-example fake-user  
kubectl create rolebinding -n psp-example fake-editor --  
clusterrole=edit --serviceaccount=psp-example:fake-user
```

To make it clear which user we're acting as and save some typing, create 2 aliases:

```
alias kubectl-admin='kubectl -n psp-example'  
alias kubectl-user='kubectl --as=system:serviceaccount:psp-  
example:fake-user -n psp-example'
```

Create a policy and a pod

Define the example PodSecurityPolicy object in a file. This is a policy that simply prevents the creation of privileged pods. The name of a PodSecurityPolicy object must be a valid [DNS subdomain name](#).

[policy/example-psp.yaml](#)



```
apiVersion: policy/v1beta1  
kind: PodSecurityPolicy  
metadata:  
  name: example  
spec:  
  privileged: false # Don't allow privileged pods!  
  # The rest fills in some required fields.  
  seLinux:  
    rule: RunAsAny  
  supplementalGroups:  
    rule: RunAsAny  
  runAsUser:  
    rule: RunAsAny  
  fsGroup:  
    rule: RunAsAny  
  volumes:  
  - '*'
```

And create it with kubectl:

```
kubectl-admin create -f example-psp.yaml
```

Now, as the unprivileged user, try to create a simple pod:

```
kubectl-user create -f- <<EOF  
apiVersion: v1
```

```
kind: Pod
metadata:
  name: pause
spec:
  containers:
    - name: pause
      image: k8s.gcr.io/pause
EOF
```

The output is similar to this:

```
Error from server (Forbidden): error when creating "STDIN": pods
"pause" is forbidden: unable to validate against any pod
security policy: []
```

What happened? Although the PodSecurityPolicy was created, neither the pod's service account nor fake-user have permission to use the new policy:

```
kubectl-user auth can-i use podsecuritypolicy/example
no
```

Create the rolebinding to grant fake-user the use verb on the example policy:

Note: This is not the recommended way! See the [next section](#) for the preferred approach.

```
kubectl-admin create role psp:unprivileged \
  --verb=use \
  --resource=podsecuritypolicy \
  --resource-name=example
role "psp:unprivileged" created

kubectl-admin create rolebinding fake-user:psp:unprivileged \
  --role=psp:unprivileged \
  --serviceaccount=psp-example:fake-user
rolebinding "fake-user:psp:unprivileged" created

kubectl-user auth can-i use podsecuritypolicy/example
yes
```

Now retry creating the pod:

```
kubectl-user create -f- <<EOF
apiVersion: v1
kind: Pod
metadata:
  name: pause
spec:
  containers:
    - name: pause
      image: k8s.gcr.io/pause
EOF
```

The output is similar to this

```
pod "pause" created
```

It works as expected! But any attempts to create a privileged pod should still be denied:

```
kubectl-user create -f- <<EOF
apiVersion: v1
kind: Pod
metadata:
  name: privileged
spec:
  containers:
    - name: pause
      image: k8s.gcr.io/pause
      securityContext:
        privileged: true
EOF
```

The output is similar to this:

```
Error from server (Forbidden): error when creating "STDIN": pods "privileged" is forbidden: unable to validate against any pod security policy: [spec.containers[0].securityContext.privileged: Invalid value: true: Privileged containers are not allowed]
```

Delete the pod before moving on:

```
kubectl-user delete pod pause
```

Run another pod

Let's try that again, slightly differently:

```
kubectl-user create deployment pause --image=k8s.gcr.io/pause
deployment "pause" created
```

```
kubectl-user get pods
No resources found.
```

```
kubectl-user get events | head -n 2
LASTSEEN   FIRSTSEEN   COUNT      NAME          KIND
SUBOBJECT           TYPE      REASON
SOURCE                MESSAGE
1m         2m          15      pause-7774d79b5
ReplicaSet           Warning
FailedCreate          replicaset-controller
Error creating: pods "pause-7774d79b5-" is forbidden: no
providers available to validate pod request
```

What happened? We already bound the `psp:unprivileged` role for our `fake-user`, why are we getting the error `Error creating: pods`

"pause-7774d79b5-" is forbidden: no providers available to validate pod request? The answer lies in the source - replicaset-controller. Fake-user successfully created the deployment (which successfully created a replicaset), but when the replicaset went to create the pod it was not authorized to use the example podsecuritypolicy.

In order to fix this, bind the psp:unprivileged role to the pod's service account instead. In this case (since we didn't specify it) the service account is default:

```
kubectl-admin create rolebinding default:psp:unprivileged \
  --role=psp:unprivileged \
  --serviceaccount=psp-example:default
rolebinding "default:psp:unprivileged" created
```

Now if you give it a minute to retry, the replicaset-controller should eventually succeed in creating the pod:

```
kubectl-user get pods --watch
NAME                  READY   STATUS    RESTARTS   AGE
pause-7774d79b5-qrgcb 0/1     Pending   0          1s
pause-7774d79b5-qrgcb 0/1     Pending   0          1s
pause-7774d79b5-qrgcb 0/1     ContainerCreating   0
1s
pause-7774d79b5-qrgcb 1/1     Running   0          2s
```

Clean up

Delete the namespace to clean up most of the example resources:

```
kubectl-admin delete ns psp-example
namespace "psp-example" deleted
```

Note that PodSecurityPolicy resources are not namespaced, and must be cleaned up separately:

```
kubectl-admin delete psp example
podsecuritypolicy "example" deleted
```

Example Policies

This is the least restrictive policy you can create, equivalent to not using the pod security policy admission controller:

[policy/privileged-psp.yaml](#)



```
apiVersion: policy/v1beta1
kind: PodSecurityPolicy
metadata:
  name: privileged
  annotations:
```

```

    seccomp.security.alpha.kubernetes.io/allowedProfileNames: '*'
spec:
  privileged: true
  allowPrivilegeEscalation: true
  allowedCapabilities:
  - '*'
  volumes:
  - '*'
  hostNetwork: true
  hostPorts:
  - min: 0
    max: 65535
  hostIPC: true
  hostPID: true
  runAsUser:
    rule: 'RunAsAny'
  seLinux:
    rule: 'RunAsAny'
  supplementalGroups:
    rule: 'RunAsAny'
  fsGroup:
    rule: 'RunAsAny'

```

This is an example of a restrictive policy that requires users to run as an unprivileged user, blocks possible escalations to root, and requires use of several security mechanisms.

[policy/restricted-psp.yaml](#)



```

apiVersion: policy/v1beta1
kind: PodSecurityPolicy
metadata:
  name: restricted
  annotations:
    seccomp.security.alpha.kubernetes.io/allowedProfileNames: 'docker/default, runtime/default'
    apparmor.security.beta.kubernetes.io/allowedProfileNames: 'runtime/default'
    seccomp.security.alpha.kubernetes.io/defaultProfileName: 'runtime/default'
    apparmor.security.beta.kubernetes.io/defaultProfileName: 'runtime/default'
spec:
  privileged: false
  # Required to prevent escalations to root.
  allowPrivilegeEscalation: false
  # This is redundant with non-root + disallow privilege escalation,
  # but we can provide it for defense in depth.
  requiredDropCapabilities:
  - ALL

```

```

# Allow core volume types.
volumes:
  - 'configMap'
  - 'emptyDir'
  - 'projected'
  - 'secret'
  - 'downwardAPI'
  # Assume that persistentVolumes set up by the cluster admin
are safe to use.
  - 'persistentVolumeClaim'
hostNetwork: false
hostIPC: false
hostPID: false
runAsUser:
  # Require the container to run without root privileges.
  rule: 'MustRunAsNonRoot'
seLinux:
  # This policy assumes the nodes are using AppArmor rather
than SELinux.
  rule: 'RunAsAny'
supplementalGroups:
  rule: 'MustRunAs'
  ranges:
    # Forbid adding the root group.
    - min: 1
      max: 65535
fsGroup:
  rule: 'MustRunAs'
  ranges:
    # Forbid adding the root group.
    - min: 1
      max: 65535
readOnlyRootFilesystem: false

```

See [Pod Security Standards](#) for more examples.

Policy Reference

Privileged

Privileged - determines if any container in a pod can enable privileged mode. By default a container is not allowed to access any devices on the host, but a "privileged" container is given access to all devices on the host. This allows the container nearly all the same access as processes running on the host. This is useful for containers that want to use linux capabilities like manipulating the network stack and accessing devices.

Host namespaces

HostPID - Controls whether the pod containers can share the host process ID namespace. Note that when paired with ptrace this can be used to escalate privileges outside of the container (ptrace is forbidden by default).

HostIPC - Controls whether the pod containers can share the host IPC namespace.

HostNetwork - Controls whether the pod may use the node network namespace. Doing so gives the pod access to the loopback device, services listening on localhost, and could be used to snoop on network activity of other pods on the same node.

HostPorts - Provides a list of ranges of allowable ports in the host network namespace. Defined as a list of `HostPortRange`, with `min(inclusive)` and `max(inclusive)`. Defaults to no allowed host ports.

Volumes and file systems

Volumes - Provides a list of allowed volume types. The allowable values correspond to the volume sources that are defined when creating a volume. For the complete list of volume types, see [Types of Volumes](#). Additionally, * may be used to allow all volume types.

The **recommended minimum set** of allowed volumes for new PSPs are:

- `configMap`
- `downwardAPI`
- `emptyDir`
- `persistentVolumeClaim`
- `secret`
- `projected`

Warning: `PodSecurityPolicy` does not limit the types of `PersistentVolume` objects that may be referenced by a `PersistentVolumeClaim`, and `hostPath` type `PersistentVolumes` do not support read-only access mode. Only trusted users should be granted permission to create `PersistentVolume` objects.

FSGroup - Controls the supplemental group applied to some volumes.

- `MustRunAs` - Requires at least one range to be specified. Uses the minimum value of the first range as the default. Validates against all ranges.
- `MayRunAs` - Requires at least one range to be specified. Allows `FSGroup`s to be left unset without providing a default. Validates against all ranges if `FSGroups` is set.
- `RunAsAny` - No default provided. Allows any `fsGroup` ID to be specified.

AllowedHostPaths - This specifies a list of host paths that are allowed to be used by `hostPath` volumes. An empty list means there is no restriction on host paths used. This is defined as a list of objects with a single `pathPrefix`

field, which allows hostPath volumes to mount a path that begins with an allowed prefix, and a readOnly field indicating it must be mounted read-only. For example:

```
allowedHostPaths:
# This allows "/foo", "/foo/", "/foo/bar" etc., but
# disallows "/fool", "/etc/foo" etc.
# "/foo/.." is never valid.
- pathPrefix: "/foo"
readOnly: true # only allow read-only mounts
```

Warning:

There are many ways a container with unrestricted access to the host filesystem can escalate privileges, including reading data from other containers, and abusing the credentials of system services, such as Kubelet.

*Writeable hostPath directory volumes allow containers to write to the filesystem in ways that let them traverse the host filesystem outside the pathPrefix. readOnly: true, available in Kubernetes 1.11+, must be used on **all** allowedHostPaths to effectively limit access to the specified pathPrefix.*

ReadOnlyRootFilesystem - Requires that containers must run with a read-only root filesystem (i.e. no writable layer).

FlexVolume drivers

This specifies a list of FlexVolume drivers that are allowed to be used by flexvolume. An empty list or nil means there is no restriction on the drivers. Please make sure [volumes](#) field contains the flexVolume volume type; no FlexVolume driver is allowed otherwise.

For example:

```
apiVersion: policy/v1beta1
kind: PodSecurityPolicy
metadata:
  name: allow-flex-volumes
spec:
  # ... other spec fields
  volumes:
    - flexVolume
  allowedFlexVolumes:
    - driver: example/lvm
    - driver: example/cifs
```

Users and groups

RunAsUser - Controls which user ID the containers are run with.

- *MustRunAs* - Requires at least one range to be specified. Uses the minimum value of the first range as the default. Validates against all ranges.
- *MustRunAsNonRoot* - Requires that the pod be submitted with a non-zero *runAsUser* or have the *USER* directive defined (using a numeric *UID*) in the image. Pods which have specified neither *runAsNonRoot* nor *runAsUser* settings will be mutated to set *runAsNonRoot=true*, thus requiring a defined non-zero numeric *USER* directive in the container. No default provided. Setting *allowPrivilegeEscalation=false* is strongly recommended with this strategy.
- *RunAsAny* - No default provided. Allows any *runAsUser* to be specified.

RunAsGroup - Controls which primary group ID the containers are run with.

- *MustRunAs* - Requires at least one range to be specified. Uses the minimum value of the first range as the default. Validates against all ranges.
- *MayRunAs* - Does not require that *RunAsGroup* be specified. However, when *RunAsGroup* is specified, they have to fall in the defined range.
- *RunAsAny* - No default provided. Allows any *runAsGroup* to be specified.

SupplementalGroups - Controls which group IDs containers add.

- *MustRunAs* - Requires at least one range to be specified. Uses the minimum value of the first range as the default. Validates against all ranges.
- *MayRunAs* - Requires at least one range to be specified. Allows *supplementalGroups* to be left unset without providing a default. Validates against all ranges if *supplementalGroups* is set.
- *RunAsAny* - No default provided. Allows any *supplementalGroups* to be specified.

Privilege Escalation

These options control the *allowPrivilegeEscalation* container option. This bool directly controls whether the [*no_new_privs*](#) flag gets set on the container process. This flag will prevent setuid binaries from changing the effective user ID, and prevent files from enabling extra capabilities (e.g. it will prevent the use of the ping tool). This behavior is required to effectively enforce *MustRunAsNonRoot*.

AllowPrivilegeEscalation - Gates whether or not a user is allowed to set the security context of a container to *allowPrivilegeEscalation=true*. This defaults to allowed so as to not break setuid binaries. Setting it to false ensures that no child process of a container can gain more privileges than its parent.

DefaultAllowPrivilegeEscalation - Sets the default for the `allowPrivilegeEscalation` option. The default behavior without this is to allow privilege escalation so as to not break setuid binaries. If that behavior is not desired, this field can be used to default to disallow, while still permitting pods to request `allowPrivilegeEscalation` explicitly.

Capabilities

Linux capabilities provide a finer grained breakdown of the privileges traditionally associated with the superuser. Some of these capabilities can be used to escalate privileges or for container breakout, and may be restricted by the PodSecurityPolicy. For more details on Linux capabilities, see [capabilities\(7\)](#).

The following fields take a list of capabilities, specified as the capability name in ALL_CAPS without the CAP_ prefix.

AllowedCapabilities - Provides a list of capabilities that are allowed to be added to a container. The default set of capabilities are implicitly allowed. The empty set means that no additional capabilities may be added beyond the default set. * can be used to allow all capabilities.

RequiredDropCapabilities - The capabilities which must be dropped from containers. These capabilities are removed from the default set, and must not be added. Capabilities listed in RequiredDropCapabilities must not be included in AllowedCapabilities or DefaultAddCapabilities.

DefaultAddCapabilities - The capabilities which are added to containers by default, in addition to the runtime defaults. See the [Docker documentation](#) for the default list of capabilities when using the Docker runtime.

SELinux

- `MustRunAs` - Requires `seLinuxOptions` to be configured. Uses `seLinuxOptions` as the default. Validates against `seLinuxOptions`.
- `RunAsAny` - No default provided. Allows any `seLinuxOptions` to be specified.

AllowedProcMountTypes

`allowedProcMountTypes` is a list of allowed `ProcMountTypes`. Empty or nil indicates that only the `DefaultProcMountType` may be used.

`DefaultProcMount` uses the container runtime defaults for readonly and masked paths for /proc. Most container runtimes mask certain paths in /proc to avoid accidental security exposure of special devices or information. This is denoted as the string `Default`.

The only other `ProcMountType` is `UnmaskedProcMount`, which bypasses the default masking behavior of the container runtime and ensures the newly

created /proc the container stays intact with no modifications. This is denoted as the string *Unmasked*.

AppArmor

Controlled via annotations on the PodSecurityPolicy. Refer to the [AppArmor documentation](#).

Seccomp

As of Kubernetes v1.19, you can use the `seccompProfile` field in the `securityContext` of Pods or containers to [control use of seccomp profiles](#). In prior versions, seccomp was controlled by adding annotations to a Pod. The same PodSecurityPolicies can be used with either version to enforce how these fields or annotations are applied.

seccomp.security.alpha.kubernetes.io/defaultProfileName - Annotation that specifies the default seccomp profile to apply to containers. Possible values are:

- `unconfined` - Seccomp is not applied to the container processes (this is the default in Kubernetes), if no alternative is provided.
- `runtime/default` - The default container runtime profile is used.
- `docker/default` - The Docker default seccomp profile is used.
Deprecated as of Kubernetes 1.11. Use `runtime/default` instead.
- `localhost/<path>` - Specify a profile as a file on the node located at `<seccomp_root>/<path>`, where `<seccomp_root>` is defined via the `--seccomp-profile-root` flag on the Kubelet. If the `--seccomp-profile-root` flag is not defined, the default path will be used, which is `<root-dir>/seccomp` where `<root-dir>` is specified by the `--root-dir` flag.

Note: The `--seccomp-profile-root` flag is deprecated since Kubernetes v1.19. Users are encouraged to use the default path.

seccomp.security.alpha.kubernetes.io/allowedProfileNames -

Annotation that specifies which values are allowed for the pod seccomp annotations. Specified as a comma-delimited list of allowed values. Possible values are those listed above, plus * to allow all profiles. Absence of this annotation means that the default cannot be changed.

Sysctl

By default, all safe sysctls are allowed.

- `forbiddenSysctls` - excludes specific sysctls. You can forbid a combination of safe and unsafe sysctls in the list. To forbid setting any sysctls, use * on its own.
- `allowedUnsafeSysctls` - allows specific sysctls that had been disallowed by the default list, so long as these are not listed in `forbiddenSysctls`.

Refer to the [Sysctl documentation](#).

What's next

- See [Pod Security Standards](#) for policy recommendations.
- Refer to [Pod Security Policy Reference](#) for the api details.

Feedback

Was this page helpful?

Yes No

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

Last modified December 02, 2020 at 3:43 PM PST: [trim whitespaces in manifests \(28c964b5c\)](#)

[Edit this page](#) [Create child page](#) [Create an issue](#)

- [What is a Pod Security Policy?](#)
- [Enabling Pod Security Policies](#)
- [Authorizing Policies](#)
 - [Via RBAC](#)
 - [Troubleshooting](#)
- [Policy Order](#)
- [Example](#)
 - [Set up](#)
 - [Create a policy and a pod](#)
 - [Run another pod](#)
 - [Clean up](#)
 - [Example Policies](#)
- [Policy Reference](#)
 - [Privileged](#)
 - [Host namespaces](#)
 - [Volumes and file systems](#)
 - [FlexVolume drivers](#)
 - [Users and groups](#)
 - [Privilege Escalation](#)
 - [Capabilities](#)
 - [SELinux](#)
 - [AllowedProcMountTypes](#)
 - [AppArmor](#)
 - [Seccomp](#)
 - [Sysctl](#)
- [What's next](#)

Process ID Limits And Reservations

FEATURE STATE: Kubernetes v1.20 [stable]

Kubernetes allow you to limit the number of process IDs (PIDs) that a [Pod](#) can use. You can also reserve a number of allocatable PIDs for each [node](#) for use by the operating system and daemons (rather than by Pods).

Process IDs (PIDs) are a fundamental resource on nodes. It is trivial to hit the task limit without hitting any other resource limits, which can then cause instability to a host machine.

Cluster administrators require mechanisms to ensure that Pods running in the cluster cannot induce PID exhaustion that prevents host daemons (such as the [kubelet](#) or [kube-proxy](#), and potentially also the container runtime) from running. In addition, it is important to ensure that PIDs are limited among Pods in order to ensure they have limited impact on other workloads on the same node.

Note: On certain Linux installations, the operating system sets the PIDs limit to a low default, such as 32768. Consider raising the value of `/proc/sys/kernel/pid_max`.

You can configure a kubelet to limit the number of PIDs a given Pod can consume. For example, if your node's host OS is set to use a maximum of 262 144 PIDs and expect to host less than 250 Pods, one can give each Pod a budget of 1000 PIDs to prevent using up that node's overall number of available PIDs. If the admin wants to overcommit PIDs similar to CPU or memory, they may do so as well with some additional risks. Either way, a single Pod will not be able to bring the whole machine down. This kind of resource limiting helps to prevent simple fork bombs from affecting operation of an entire cluster.

Per-Pod PID limiting allows administrators to protect one Pod from another, but does not ensure that all Pods scheduled onto that host are unable to impact the node overall. Per-Pod limiting also does not protect the node agents themselves from PID exhaustion.

You can also reserve an amount of PIDs for node overhead, separate from the allocation to Pods. This is similar to how you can reserve CPU, memory, or other resources for use by the operating system and other facilities outside of Pods and their containers.

PID limiting is a sibling to [compute resource](#) requests and limits. However, you specify it in a different way: rather than defining a Pod's resource limit in the `.spec` for a Pod, you configure the limit as a setting on the kubelet. Pod-defined PID limits are not currently supported.

Caution: This means that the limit that applies to a Pod may be different depending on where the Pod is scheduled. To make things

simple, it's easiest if all Nodes use the same PID resource limits and reservations.

Node PID limits

Kubernetes allows you to reserve a number of process IDs for the system use. To configure the reservation, use the parameter `pid=<number>` in the `--system-reserved` and `--kube-reserved` command line options to the kubelet. The value you specified declares that the specified number of process IDs will be reserved for the system as a whole and for Kubernetes system daemons respectively.

Note: Before Kubernetes version 1.20, PID resource limiting with Node-level reservations required enabling the [feature gate](#) `SupportNodePidsLimit` to work.

Pod PID limits

Kubernetes allows you to limit the number of processes running in a Pod. You specify this limit at the node level, rather than configuring it as a resource limit for a particular Pod. Each Node can have a different PID limit. To configure the limit, you can specify the command line parameter `--pod-max-pids` to the kubelet, or set `PodPidsLimit` in the kubelet [configuration file](#).

Note: Before Kubernetes version 1.20, PID resource limiting for Pods required enabling the [feature gate](#) `SupportPodPidsLimit` to work.

PID based eviction

You can configure kubelet to start terminating a Pod when it is misbehaving and consuming abnormal amount of resources. This feature is called eviction. You can [Configure Out of Resource Handling](#) for various eviction signals. Use `pid.available` eviction signal to configure the threshold for number of PIDs used by Pod. You can set soft and hard eviction policies. However, even with the hard eviction policy, if the number of PIDs growing very fast, node can still get into unstable state by hitting the node PIDs limit. Eviction signal value is calculated periodically and does NOT enforce the limit.

PID limiting - per Pod and per Node sets the hard limit. Once the limit is hit, workload will start experiencing failures when trying to get a new PID. It may or may not lead to rescheduling of a Pod, depending on how workload reacts on these failures and how liveness and readiness probes are configured for the Pod. However, if limits were set correctly, you can guarantee that other Pods workload and system processes will not run out of PIDs when one Pod is misbehaving.

What's next

- Refer to the [PID Limiting enhancement document](#) for more information.
- For historical context, read [Process ID Limiting for Stability Improvements in Kubernetes 1.14](#).
- Read [Managing Resources for Containers](#).
- Learn how to [Configure Out of Resource Handling](#).

Feedback

Was this page helpful?

Yes No

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

Last modified December 08, 2020 at 10:58 PM PST: [add `pid.available` to the eviction signals list \(d1dc73cb3\)](#)

[Edit this page](#) [Create child page](#) [Create an issue](#)

- [Node PID limits](#)
- [Pod PID limits](#)
- [PID based eviction](#)
- [What's next](#)

Scheduling and Eviction

In Kubernetes, scheduling refers to making sure that Pods are matched to Nodes so that the kubelet can run them. Eviction is the process of proactively failing one or more Pods on resource-starved Nodes.

[Kubernetes Scheduler](#)

[Taints and Tolerations](#)

[Assigning Pods to Nodes](#)

[Pod Overhead](#)

[Resource Bin Packing for Extended Resources](#)

[Eviction Policy](#)

[Scheduling Framework](#)

[Scheduler Performance Tuning](#)

Kubernetes Scheduler

In Kubernetes, scheduling refers to making sure that [Pods](#) are matched to [Nodes](#) so that [Kubelet](#) can run them.

Scheduling overview

A scheduler watches for newly created Pods that have no Node assigned. For every Pod that the scheduler discovers, the scheduler becomes responsible for finding the best Node for that Pod to run on. The scheduler reaches this placement decision taking into account the scheduling principles described below.

If you want to understand why Pods are placed onto a particular Node, or if you're planning to implement a custom scheduler yourself, this page will help you learn about scheduling.

kube-scheduler

[kube-scheduler](#) is the default scheduler for Kubernetes and runs as part of the [control plane](#). kube-scheduler is designed so that, if you want and need to, you can write your own scheduling component and use that instead.

For every newly created pod or other unscheduled pods, kube-scheduler selects an optimal node for them to run on. However, every container in pods has different requirements for resources and every pod also has different requirements. Therefore, existing nodes need to be filtered according to the specific scheduling requirements.

In a cluster, Nodes that meet the scheduling requirements for a Pod are called **feasible nodes**. If none of the nodes are suitable, the pod remains unscheduled until the scheduler is able to place it.

The scheduler finds feasible Nodes for a Pod and then runs a set of functions to score the feasible Nodes and picks a Node with the highest score among the feasible ones to run the Pod. The scheduler then notifies the API server about this decision in a process called **binding**.

Factors that need taken into account for scheduling decisions include individual and collective resource requirements, hardware / software / policy constraints, affinity and anti-affinity specifications, data locality, inter-workload interference, and so on.

Node selection in kube-scheduler

`kube-scheduler` selects a node for the pod in a 2-step operation:

1. Filtering
2. Scoring

The filtering step finds the set of Nodes where it's feasible to schedule the Pod. For example, the `PodFitsResources` filter checks whether a candidate Node has enough available resource to meet a Pod's specific resource requests. After this step, the node list contains any suitable Nodes; often, there will be more than one. If the list is empty, that Pod isn't (yet) schedulable.

In the scoring step, the scheduler ranks the remaining nodes to choose the most suitable Pod placement. The scheduler assigns a score to each Node that survived filtering, basing this score on the active scoring rules.

Finally, `kube-scheduler` assigns the Pod to the Node with the highest ranking. If there is more than one node with equal scores, `kube-scheduler` selects one of these at random.

There are two supported ways to configure the filtering and scoring behavior of the scheduler:

1. [Scheduling Policies](#) allow you to configure Predicates for filtering and Priorities for scoring.
2. [Scheduling Profiles](#) allow you to configure Plugins that implement different scheduling stages, including: `QueueSort`, `Filter`, `Score`, `Bind`, `Reserve`, `Permit`, and others. You can also configure the `kube-scheduler` to run different profiles.

What's next

- Read about [scheduler performance tuning](#)
- Read about [Pod topology spread constraints](#)
- Read the [reference documentation](#) for `kube-scheduler`
- Learn about [configuring multiple schedulers](#)
- Learn about [topology management policies](#)
- Learn about [Pod Overhead](#)
- Learn about scheduling of Pods that use volumes in:
 - [Volume Topology Support](#)
 - [Storage Capacity Tracking](#)
 - [Node-specific Volume Limits](#)

Feedback

Was this page helpful?

Yes No

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

Last modified October 22, 2020 at 2:24 PM PST: [Fix links in concepts section \(070023b24\)](#)

- [Scheduling overview](#)
- [kube-scheduler](#)
 - [Node selection in kube-scheduler](#)
- [What's next](#)

Taints and Tolerations

[Node affinity](#), is a property of [Pods](#) that attracts them to a set of [nodes](#) (either as a preference or a hard requirement). Taints are the opposite -- they allow a node to repel a set of pods.

Tolerations are applied to pods, and allow (but do not require) the pods to schedule onto nodes with matching taints.

Taints and tolerations work together to ensure that pods are not scheduled onto inappropriate nodes. One or more taints are applied to a node; this marks that the node should not accept any pods that do not tolerate the taints.

Concepts

You add a taint to a node using [kubectl taint](#). For example,

```
kubectl taint nodes node1 key1=value1:NoSchedule
```

places a taint on node `node1`. The taint has key `key1`, value `value1`, and taint effect `NoSchedule`. This means that no pod will be able to schedule onto node `node1` unless it has a matching toleration.

To remove the taint added by the command above, you can run:

```
kubectl taint nodes node1 key1=value1:NoSchedule-
```

You specify a toleration for a pod in the PodSpec. Both of the following tolerations "match" the taint created by the `kubectl taint` line above, and thus a pod with either toleration would be able to schedule onto `node1`:

```
tolerations:  
- key: "key1"  
  operator: "Equal"  
  value: "value1"  
  effect: "NoSchedule"
```

```
tolerations:  
- key: "key1"  
  operator: "Exists"  
  effect: "NoSchedule"
```

Here's an example of a pod that uses tolerations:



```
apiVersion: v1
kind: Pod
metadata:
  name: nginx
  labels:
    env: test
spec:
  containers:
    - name: nginx
      image: nginx
      imagePullPolicy: IfNotPresent
  tolerations:
    - key: "example-key"
      operator: "Exists"
      effect: "NoSchedule"
```

The default value for `operator` is `Equal`.

A toleration "matches" a taint if the keys are the same and the effects are the same, and:

- the `operator` is `Exists` (in which case no `value` should be specified), or
- the `operator` is `Equal` and the `values` are equal.

Note:

There are two special cases:

An empty `key` with `operator Exists` matches all keys, values and effects which means this will tolerate everything.

An empty `effect` matches all effects with key `key1`.

The above example used `effect` of `NoSchedule`. Alternatively, you can use `effect` of `PreferNoSchedule`. This is a "preference" or "soft" version of `NoSchedule` -- the system will try to avoid placing a pod that does not tolerate the taint on the node, but it is not required. The third kind of effect is `NoExecute`, described later.

You can put multiple taints on the same node and multiple tolerations on the same pod. The way Kubernetes processes multiple taints and tolerations is like a filter: start with all of a node's taints, then ignore the ones for which the pod has a matching toleration; the remaining un-ignored taints have the indicated effects on the pod. In particular,

- if there is at least one un-ignored taint with effect `NoSchedule` then Kubernetes will not schedule the pod onto that node

- if there is no un-ignored taint with effect `NoSchedule` but there is at least one un-ignored taint with effect `PreferNoSchedule` then Kubernetes will try to not schedule the pod onto the node
- if there is at least one un-ignored taint with effect `NoExecute` then the pod will be evicted from the node (if it is already running on the node), and will not be scheduled onto the node (if it is not yet running on the node).

For example, imagine you taint a node like this

```
kubectl taint nodes node1 key1=value1:NoSchedule
kubectl taint nodes node1 key1=value1:NoExecute
kubectl taint nodes node1 key2=value2:NoSchedule
```

And a pod has two tolerations:

```
tolerations:
- key: "key1"
  operator: "Equal"
  value: "value1"
  effect: "NoSchedule"
- key: "key1"
  operator: "Equal"
  value: "value1"
  effect: "NoExecute"
```

In this case, the pod will not be able to schedule onto the node, because there is no toleration matching the third taint. But it will be able to continue running if it is already running on the node when the taint is added, because the third taint is the only one of the three that is not tolerated by the pod.

Normally, if a taint with effect `NoExecute` is added to a node, then any pods that do not tolerate the taint will be evicted immediately, and pods that do tolerate the taint will never be evicted. However, a toleration with `NoExecute` effect can specify an optional `tolerationSeconds` field that dictates how long the pod will stay bound to the node after the taint is added. For example,

```
tolerations:
- key: "key1"
  operator: "Equal"
  value: "value1"
  effect: "NoExecute"
  tolerationSeconds: 3600
```

means that if this pod is running and a matching taint is added to the node, then the pod will stay bound to the node for 3600 seconds, and then be evicted. If the taint is removed before that time, the pod will not be evicted.

Example Use Cases

Taints and tolerations are a flexible way to steer pods away from nodes or evict pods that shouldn't be running. A few of the use cases are

- **Dedicated Nodes:** If you want to dedicate a set of nodes for exclusive use by a particular set of users, you can add a taint to those nodes (say, `kubectl taint nodes nodename dedicated=groupName:NoSchedule`) and then add a corresponding toleration to their pods (this would be done most easily by writing a custom [admission controller](#)). The pods with the tolerations will then be allowed to use the tainted (dedicated) nodes as well as any other nodes in the cluster. If you want to dedicate the nodes to them and ensure they only use the dedicated nodes, then you should additionally add a label similar to the taint to the same set of nodes (e.g. `dedicated=groupName`), and the admission controller should additionally add a node affinity to require that the pods can only schedule onto nodes labeled with `dedicated=groupName`.
- **Nodes with Special Hardware:** In a cluster where a small subset of nodes have specialized hardware (for example GPUs), it is desirable to keep pods that don't need the specialized hardware off of those nodes, thus leaving room for later-arriving pods that do need the specialized hardware. This can be done by tainting the nodes that have the specialized hardware (e.g. `kubectl taint nodes nodename special=true:NoSchedule` or `kubectl taint nodes nodename special=true:PreferNoSchedule`) and adding a corresponding toleration to pods that use the special hardware. As in the dedicated nodes use case, it is probably easiest to apply the tolerations using a custom [admission controller](#). For example, it is recommended to use [Extended Resources](#) to represent the special hardware, taint your special hardware nodes with the extended resource name and run the [ExtendedResourceToleration](#) admission controller. Now, because the nodes are tainted, no pods without the toleration will schedule on them. But when you submit a pod that requests the extended resource, the `ExtendedResourceToleration` admission controller will automatically add the correct toleration to the pod and that pod will schedule on the special hardware nodes. This will make sure that these special hardware nodes are dedicated for pods requesting such hardware and you don't have to manually add tolerations to your pods.
- **Taint based Evictions:** A per-pod-configurable eviction behavior when there are node problems, which is described in the next section.

Taint based Evictions

FEATURE STATE: Kubernetes v1.18 [stable]

The `NoExecute` taint effect, mentioned above, affects pods that are already running on the node as follows

- pods that do not tolerate the taint are evicted immediately

- pods that tolerate the taint without specifying `tolerationSeconds` in their toleration specification remain bound forever
- pods that tolerate the taint with a specified `tolerationSeconds` remain bound for the specified amount of time

The node controller automatically taints a Node when certain conditions are true. The following taints are built in:

- `node.kubernetes.io/not-ready`: Node is not ready. This corresponds to the `NodeCondition Ready` being "False".
- `node.kubernetes.io/unreachable`: Node is unreachable from the node controller. This corresponds to the `NodeCondition Ready` being "Unknown".
- `node.kubernetes.io/out-of-disk`: Node becomes out of disk.
- `node.kubernetes.io/memory-pressure`: Node has memory pressure.
- `node.kubernetes.io/disk-pressure`: Node has disk pressure.
- `node.kubernetes.io/network-unavailable`: Node's network is unavailable.
- `node.kubernetes.io/unschedulable`: Node is unschedulable.
- `node.cloudprovider.kubernetes.io/uninitialized`: When the kubelet is started with "external" cloud provider, this taint is set on a node to mark it as unusable. After a controller from the cloud-controller-manager initializes this node, the kubelet removes this taint.

In case a node is to be evicted, the node controller or the kubelet adds relevant taints with `NoExecute` effect. If the fault condition returns to normal the kubelet or node controller can remove the relevant taint(s).

Note: The control plane limits the rate of adding new taints to nodes. This rate limiting manages the number of evictions that are triggered when many nodes become unreachable at once (for example: if there is a network disruption).

You can specify `tolerationSeconds` for a Pod to define how long that Pod stays bound to a failing or unresponsive Node.

For example, you might want to keep an application with a lot of local state bound to node for a long time in the event of network partition, hoping that the partition will recover and thus the pod eviction can be avoided. The toleration you set for that Pod might look like:

```
tolerations:
- key: "node.kubernetes.io/unreachable"
  operator: "Exists"
  effect: "NoExecute"
  tolerationSeconds: 6000
```

Note:

Kubernetes automatically adds a toleration for `node.kubernetes.io/not-ready` and `node.kubernetes.io/unreachable` with `tolerationSeconds=300`, unless you, or a controller, set those tolerations explicitly.

These automatically-added tolerations mean that Pods remain bound to Nodes for 5 minutes after one of these problems is detected.

[DaemonSet](#) pods are created with `NoExecute` tolerations for the following taints with no `tolerationSeconds`:

- `node.kubernetes.io/unreachable`
- `node.kubernetes.io/not-ready`

This ensures that DaemonSet pods are never evicted due to these problems.

Taint Nodes by Condition

The node lifecycle controller automatically creates taints corresponding to Node conditions with `NoSchedule` effect. Similarly the scheduler does not check Node conditions; instead the scheduler checks taints. This assures that Node conditions don't affect what's scheduled onto the Node. The user can choose to ignore some of the Node's problems (represented as Node conditions) by adding appropriate Pod tolerations.

The DaemonSet controller automatically adds the following `NoSchedule` tolerations to all daemons, to prevent DaemonSets from breaking.

- `node.kubernetes.io/memory-pressure`
- `node.kubernetes.io/disk-pressure`
- `node.kubernetes.io/out-of-disk` (only for critical pods)
- `node.kubernetes.io/unschedulable` (1.10 or later)
- `node.kubernetes.io/network-unavailable` (host network only)

Adding these tolerations ensures backward compatibility. You can also add arbitrary tolerations to DaemonSets.

What's next

- Read about [out of resource handling](#) and how you can configure it
- Read about [pod priority](#)

Feedback

Was this page helpful?

Yes No

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

Last modified November 19, 2020 at 10:17 PM PST: [Improve Taints and Toleration example \(937ddcea5\)](#)

[Edit this page](#) [Create child page](#) [Create an issue](#)

- [Concepts](#)
- [Example Use Cases](#)
- [Taint based Evictions](#)
- [Taint Nodes by Condition](#)
- [What's next](#)

Assigning Pods to Nodes

You can constrain a [Pod](#) to only be able to run on particular [Node\(s\)](#), or to prefer to run on particular nodes. There are several ways to do this, and the recommended approaches all use [label selectors](#) to make the selection.

Generally such constraints are unnecessary, as the scheduler will automatically do a reasonable placement (e.g. spread your pods across nodes, not place the pod on a node with insufficient free resources, etc.) but there are some circumstances where you may want more control on a node where a pod lands, for example to ensure that a pod ends up on a machine with an SSD attached to it, or to co-locate pods from two different services that communicate a lot into the same availability zone.

nodeSelector

`nodeSelector` is the simplest recommended form of node selection constraint. `nodeSelector` is a field of `PodSpec`. It specifies a map of key-value pairs. For the pod to be eligible to run on a node, the node must have each of the indicated key-value pairs as labels (it can have additional labels as well). The most common usage is one key-value pair.

Let's walk through an example of how to use `nodeSelector`.

Step Zero: Prerequisites

This example assumes that you have a basic understanding of Kubernetes pods and that you have [set up a Kubernetes cluster](#).

Step One: Attach label to the node

Run `kubectl get nodes` to get the names of your cluster's nodes. Pick out the one that you want to add a label to, and then run `kubectl label nodes <node-name> <label-key>=<label-value>` to add a label to the node you've chosen. For example, if my node name is 'kubernetes-foo-node-1.c.a-robinson.internal' and my desired label is 'disktype:ssd', then I can run `kubectl label nodes kubernetes-foo-node-1.c.a-robinson.internal disktype:ssd`.

You can verify that it worked by re-running `kubectl get nodes --show-labels` and checking that the node now has a label. You can also use `kubectl describe node "nodename"` to see the full list of labels of the given node.

Step Two: Add a `nodeSelector` field to your pod configuration

Take whatever pod config file you want to run, and add a `nodeSelector` section to it, like this. For example, if this is my pod config:

```
apiVersion: v1
kind: Pod
metadata:
  name: nginx
  labels:
    env: test
spec:
  containers:
  - name: nginx
    image: nginx
```

Then add a `nodeSelector` like so:

[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
```

When you then run `kubectl apply -f https://k8s.io/examples/pods/pod-nginx.yaml`, the Pod will get scheduled on the node that you attached the label to. You can verify that it worked by running `kubectl get pods -o wide` and looking at the "NODE" that the Pod was assigned to.

Interlude: built-in node labels

In addition to labels you [attach](#), nodes come pre-populated with a standard set of labels. See [Well-Known Labels, Annotations and Taints](#) for a list of these.

Note: The value of these labels is cloud provider specific and is not guaranteed to be reliable. For example, the value of `kubernetes.io/hostname` may be the same as the Node name in some environments and a different value in other environments.

Node isolation/restriction

Adding labels to Node objects allows targeting pods to specific nodes or groups of nodes. This can be used to ensure specific pods only run on nodes with certain isolation, security, or regulatory properties. When using labels for this purpose, choosing label keys that cannot be modified by the kubelet process on the node is strongly recommended. This prevents a compromised node from using its kubelet credential to set those labels on its own Node object, and influencing the scheduler to schedule workloads to the compromised node.

The `NodeRestriction` admission plugin prevents kubelets from setting or modifying labels with a `node-restriction.kubernetes.io/` prefix. To make use of that label prefix for node isolation:

1. Ensure you are using the [Node authorizer](#) and have enabled the [NodeRestriction admission plugin](#).
2. Add labels under the `node-restriction.kubernetes.io/` prefix to your Node objects, and use those labels in your node selectors. For example, `example.com.node-restriction.kubernetes.io/fips=true` or `example.com.node-restriction.kubernetes.io/pci-dss=true`.

Affinity and anti-affinity

`nodeSelector` provides a very simple way to constrain pods to nodes with particular labels. The affinity/anti-affinity feature, greatly expands the types of constraints you can express. The key enhancements are

1. The affinity/anti-affinity language is more expressive. The language offers more matching rules besides exact matches created with a logical AND operation;
2. you can indicate that the rule is "soft"/"preference" rather than a hard requirement, so if the scheduler can't satisfy it, the pod will still be scheduled;
3. you can constrain against labels on other pods running on the node (or other topological domain), rather than against labels on the node itself, which allows rules about which pods can and cannot be co-located

The affinity feature consists of two types of affinity, "node affinity" and "inter-pod affinity/anti-affinity". Node affinity is like the existing `nodeSelector` or (but with the first two benefits listed above), while inter-pod affinity/anti-affinity constrains against pod labels rather than node labels, as described in the third item listed above, in addition to having the first and second properties listed above.

Node affinity

Node affinity is conceptually similar to nodeSelector -- it allows you to constrain which nodes your pod is eligible to be scheduled on, based on labels on the node.

There are currently two types of node affinity, called requiredDuringSchedulingIgnoredDuringExecution and preferredDuringSchedulingIgnoredDuringExecution. You can think of them as "hard" and "soft" respectively, in the sense that the former specifies rules that must be met for a pod to be scheduled onto a node (just like nodeSelector but using a more expressive syntax), while the latter specifies preferences that the scheduler will try to enforce but will not guarantee. The "IgnoredDuringExecution" part of the names means that, similar to how nodeSelector works, if labels on a node change at runtime such that the affinity rules on a pod are no longer met, the pod will still continue to run on the node. In the future we plan to offer requiredDuringSchedulingRequiredDuringExecution which will be just like requiredDuringSchedulingIgnoredDuringExecution except that it will evict pods from nodes that cease to satisfy the pods' node affinity requirements.

Thus an example of requiredDuringSchedulingIgnoredDuringExecution would be "only run the pod on nodes with Intel CPUs" and an example preferredDuringSchedulingIgnoredDuringExecution would be "try to run this set of pods in failure zone XYZ, but if it's not possible, then allow some to run elsewhere".

Node affinity is specified as field nodeAffinity of field affinity in the PodSpec.

Here's an example of a pod that uses node affinity:

[pods/pod-with-node-affinity.yaml](#)



```
apiVersion: v1
kind: Pod
metadata:
  name: with-node-affinity
spec:
  affinity:
    nodeAffinity:
      requiredDuringSchedulingIgnoredDuringExecution:
        nodeSelectorTerms:
        - matchExpressions:
          - key: kubernetes.io/e2e-az-name
            operator: In
            values:
            - e2e-az1
            - e2e-az2
      preferredDuringSchedulingIgnoredDuringExecution:
      - weight: 1
```

```


```

preference:
 matchExpressions:
 - key: another-node-label-key
 operator: In
 values:
 - another-node-label-value
containers:
 - name: with-node-affinity
 image: k8s.gcr.io/pause:2.0

```


```

This node affinity rule says the pod can only be placed on a node with a label whose key is `kubernetes.io/e2e-az-name` and whose value is either `e2e-az1` or `e2e-az2`. In addition, among nodes that meet that criteria, nodes with a label whose key is `another-node-label-key` and whose value is `another-node-label-value` should be preferred.

You can see the operator `In` being used in the example. The new node affinity syntax supports the following operators: `In`, `NotIn`, `Exists`, `DoesNotExist`, `Gt`, `Lt`. You can use `NotIn` and `DoesNotExist` to achieve node anti-affinity behavior, or use [node taints](#) to repel pods from specific nodes.

If you specify both `nodeSelector` and `nodeAffinity`, both must be satisfied for the pod to be scheduled onto a candidate node.

*If you specify multiple `nodeSelectorTerms` associated with `nodeAffinity` types, then the pod can be scheduled onto a node **if one of the** `nodeSelectorTerms` can be satisfied.*

*If you specify multiple `matchExpressions` associated with `nodeSelectorTerms`, then the pod can be scheduled onto a node **only if all** `matchExpressions` is satisfied.*

If you remove or change the label of the node where the pod is scheduled, the pod won't be removed. In other words, the affinity selection works only at the time of scheduling the pod.

The weight field in `preferredDuringSchedulingIgnoredDuringExecution` is in the range 1-100. For each node that meets all of the scheduling requirements (resource request, `RequiredDuringScheduling` affinity expressions, etc.), the scheduler will compute a sum by iterating through the elements of this field and adding "weight" to the sum if the node matches the corresponding `MatchExpressions`. This score is then combined with the scores of other priority functions for the node. The node(s) with the highest total score are the most preferred.

Node affinity per scheduling profile

FEATURE STATE: Kubernetes v1.20 [beta]

When configuring multiple [scheduling profiles](#), you can associate a profile with a Node affinity, which is useful if a profile only applies to a specific set of Nodes. To do so, add an `addedAffinity` to the args of the [NodeAffinity plugin](#) in the [scheduler configuration](#). For example:

```

apiVersion: kubescheduler.config.k8s.io/v1beta1
kind: KubeSchedulerConfiguration

profiles:
  - schedulerName: default-scheduler
  - schedulerName: foo-scheduler
    pluginConfig:
      - name: NodeAffinity
        args:
          addedAffinity:
            requiredDuringSchedulingIgnoredDuringExecution:
              nodeSelectorTerms:
                - matchExpressions:
                    - key: scheduler-profile
                      operator: In
                      values:
                        - foo

```

The `addedAffinity` is applied to all Pods that set `.spec.schedulerName` to `foo-scheduler`, in addition to the `NodeAffinity` specified in the `PodSpec`. That is, in order to match the Pod, Nodes need to satisfy `addedAffinity` and the Pod's `.spec.NodeAffinity`.

Since the `addedAffinity` is not visible to end users, its behavior might be unexpected to them. We recommend to use node labels that have clear correlation with the profile's scheduler name.

Note: The DaemonSet controller, which [creates Pods for DaemonSets](#) is not aware of scheduling profiles. For this reason, it is recommended that you keep a scheduler profile, such as the `default-scheduler`, without any `addedAffinity`. Then, the Daemonset's Pod template should use this scheduler name. Otherwise, some Pods created by the Daemonset controller might remain unschedulable.

Inter-pod affinity and anti-affinity

Inter-pod affinity and anti-affinity allow you to constrain which nodes your pod is eligible to be scheduled based on labels on pods that are already running on the node rather than based on labels on nodes. The rules are of the form "this pod should (or, in the case of anti-affinity, should not) run in an X if that X is already running one or more pods that meet rule Y". Y is expressed as a LabelSelector with an optional associated list of namespaces; unlike nodes, because pods are namespaced (and therefore the labels on pods are implicitly namespaced), a label selector over pod labels must specify which namespaces the selector should apply to. Conceptually X is a topology domain like node, rack, cloud provider zone, cloud provider region, etc. You express it using a `topologyKey` which is the key for the node label that the system uses to denote such a topology domain; for example, see the label keys listed above in the section [Interlude: built-in node labels](#).

Note: Inter-pod affinity and anti-affinity require substantial amount of processing which can slow down scheduling in large clusters significantly. We do not recommend using them in clusters larger than several hundred nodes.

Note: Pod anti-affinity requires nodes to be consistently labelled, in other words every node in the cluster must have an appropriate label matching `topologyKey`. If some or all nodes are missing the specified `topologyKey` label, it can lead to unintended behavior.

As with node affinity, there are currently two types of pod affinity and anti-affinity, called `requiredDuringSchedulingIgnoredDuringExecution` and `preferredDuringSchedulingIgnoredDuringExecution` which denote "hard" vs. "soft" requirements. See the description in the node affinity section earlier. An example of `requiredDuringSchedulingIgnoredDuringExecution` affinity would be "co-locate the pods of service A and service B in the same zone, since they communicate a lot with each other" and an example `preferredDuringSchedulingIgnoredDuringExecution` anti-affinity would be "spread the pods from this service across zones" (a hard requirement wouldn't make sense, since you probably have more pods than zones).

Inter-pod affinity is specified as field `podAffinity` of field `affinity` in the `PodSpec`. And inter-pod anti-affinity is specified as field `podAntiAffinity` of field `affinity` in the `PodSpec`.

An example of a pod that uses pod affinity:

[pods/pod-with-pod-affinity.yaml](#)



```
apiVersion: v1
kind: Pod
metadata:
  name: with-pod-affinity
spec:
  affinity:
    podAffinity:
      requiredDuringSchedulingIgnoredDuringExecution:
        - labelSelector:
            matchExpressions:
              - key: security
                operator: In
                values:
                  - S1
            topologyKey: topology.kubernetes.io/zone
      podAntiAffinity:
        preferredDuringSchedulingIgnoredDuringExecution:
          - weight: 100
            podAffinityTerm:
              labelSelector:
                matchExpressions:
```

```

    - key: security
      operator: In
      values:
        - S2
    topologyKey: topology.kubernetes.io/zone
containers:
  - name: with-pod-affinity
    image: k8s.gcr.io/pause:2.0
  
```

The affinity on this pod defines one pod affinity rule and one pod anti-affinity rule. In this example, the `podAffinity` is `requiredDuringSchedulingIgnoredDuringExecution` while the `podAntiAffinity` is `preferredDuringSchedulingIgnoredDuringExecution`. The pod affinity rule says that the pod can be scheduled onto a node only if that node is in the same zone as at least one already-running pod that has a label with key "security" and value "S1". (More precisely, the pod is eligible to run on node N if node N has a label with key `topology.kubernetes.io/zone` and some value V such that there is at least one node in the cluster with key `topology.kubernetes.io/zone` and value V that is running a pod that has a label with key "security" and value "S1".) The pod anti-affinity rule says that the pod cannot be scheduled onto a node if that node is in the same zone as a pod with label having key "security" and value "S2". See the [design doc](#) for many more examples of pod affinity and anti-affinity, both the `requiredDuringSchedulingIgnoredDuringExecution` flavor and the `preferredDuringSchedulingIgnoredDuringExecution` flavor.

The legal operators for pod affinity and anti-affinity are `In`, `NotIn`, `Exists`, `DoesNotExist`.

In principle, the `topologyKey` can be any legal label-key. However, for performance and security reasons, there are some constraints on `topologyKey`:

1. For pod affinity, empty `topologyKey` is not allowed in both `requiredDuringSchedulingIgnoredDuringExecution` and `preferredDuringSchedulingIgnoredDuringExecution`.
2. For pod anti-affinity, empty `topologyKey` is also not allowed in both `requiredDuringSchedulingIgnoredDuringExecution` and `preferredDuringSchedulingIgnoredDuringExecution`.
3. For `requiredDuringSchedulingIgnoredDuringExecution` pod anti-affinity, the admission controller `LimitPodHardAntiAffinityTopology` was introduced to limit `topologyKey` to `kubernetes.io/hostname`. If you want to make it available for custom topologies, you may modify the admission controller, or simply disable it.
4. Except for the above cases, the `topologyKey` can be any legal label-key.

In addition to `labelSelector` and `topologyKey`, you can optionally specify a list namespaces of namespaces which the `labelSelector` should match against (this goes at the same level of the definition as `labelSelector` and `topologyKey`). If omitted or empty, it defaults to the namespace of the pod where the affinity/anti-affinity definition appears.

All matchExpressions associated with requiredDuringSchedulingIgnoredDuringExecution affinity and anti-affinity must be satisfied for the pod to be scheduled onto a node.

More Practical Use-cases

Interpod Affinity and AntiAffinity can be even more useful when they are used with higher level collections such as ReplicaSets, StatefulSets, Deployments, etc. One can easily configure that a set of workloads should be co-located in the same defined topology, eg., the same node.

Always co-located in the same node

In a three node cluster, a web application has in-memory cache such as redis. We want the web-servers to be co-located with the cache as much as possible.

Here is the yaml snippet of a simple redis deployment with three replicas and selector label app=store. The deployment has PodAntiAffinity configured to ensure the scheduler does not co-locate replicas on a single node.

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: redis-cache
spec:
  selector:
    matchLabels:
      app: store
  replicas: 3
  template:
    metadata:
      labels:
        app: store
    spec:
      affinity:
        podAntiAffinity:
          requiredDuringSchedulingIgnoredDuringExecution:
            - labelSelector:
                matchExpressions:
                  - key: app
                    operator: In
                    values:
                      - store
            topologyKey: "kubernetes.io/hostname"
  containers:
    - name: redis-server
      image: redis:3.2-alpine
```

The below yaml snippet of the webserver deployment has podAntiAffinity and podAffinity configured. This informs the scheduler that all its replicas

are to be co-located with pods that have selector label `app=store`. This will also ensure that each web-server replica does not co-locate on a single node.

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: web-server
spec:
  selector:
    matchLabels:
      app: web-store
  replicas: 3
  template:
    metadata:
      labels:
        app: web-store
  spec:
    affinity:
      podAntiAffinity:
        requiredDuringSchedulingIgnoredDuringExecution:
        - labelSelector:
            matchExpressions:
            - key: app
              operator: In
              values:
              - web-store
            topologyKey: "kubernetes.io/hostname"
      podAffinity:
        requiredDuringSchedulingIgnoredDuringExecution:
        - labelSelector:
            matchExpressions:
            - key: app
              operator: In
              values:
              - store
            topologyKey: "kubernetes.io/hostname"
    containers:
    - name: web-app
      image: nginx:1.16-alpine
```

If we create the above two deployments, our three node cluster should look like below.

node-1	node-2	node-3
webserver-1	webserver-2	webserver-3
cache-1	cache-2	cache-3

As you can see, all the 3 replicas of the `web-server` are automatically co-located with the `cache` as expected.

```
kubectl get pods -o wide
```

The output is similar to this:

NAME	READY	STATUS	RESTARTS
AGE	IP	NODE	
redis-cache-1450370735-6dzlj	1/1	Running	0
8m	10.192.4.2	kube-node-3	
redis-cache-1450370735-j2j96	1/1	Running	0
8m	10.192.2.2	kube-node-1	
redis-cache-1450370735-z73mh	1/1	Running	0
8m	10.192.3.1	kube-node-2	
web-server-1287567482-5d4dz	1/1	Running	0
7m	10.192.2.3	kube-node-1	
web-server-1287567482-6f7v5	1/1	Running	0
7m	10.192.4.3	kube-node-3	
web-server-1287567482-s330j	1/1	Running	0
7m	10.192.3.2	kube-node-2	

Never co-located in the same node

The above example uses `PodAntiAffinity` rule with `topologyKey: "kubernetes.io/hostname"` to deploy the redis cluster so that no two instances are located on the same host. See [ZooKeeper tutorial](#) for an example of a `StatefulSet` configured with anti-affinity for high availability, using the same technique.

nodeName

`nodeName` is the simplest form of node selection constraint, but due to its limitations it is typically not used. `nodeName` is a field of `PodSpec`. If it is non-empty, the scheduler ignores the pod and the kubelet running on the named node tries to run the pod. Thus, if `nodeName` is provided in the `PodSpec`, it takes precedence over the above methods for node selection.

Some of the limitations of using `nodeName` to select nodes are:

- If the named node does not exist, the pod will not be run, and in some cases may be automatically deleted.
- If the named node does not have the resources to accommodate the pod, the pod will fail and its reason will indicate why, for example `OutOfMemory` or `OutOfCPU`.
- Node names in cloud environments are not always predictable or stable.

Here is an example of a pod config file using the `nodeName` field:

```
apiVersion: v1
kind: Pod
metadata:
  name: nginx
spec:
  containers:
    - name: nginx
```

```
image: nginx
nodeName: kube-01
```

The above pod will run on the node `kube-01`.

What's next

[Taints](#) allow a Node to repel a set of Pods.

The design documents for [node affinity](#) and for [inter-pod affinity/anti-affinity](#) contain extra background information about these features.

Once a Pod is assigned to a Node, the kubelet runs the Pod and allocates node-local resources. The [topology manager](#) can take part in node-level resource allocation decisions.

Feedback

Was this page helpful?

Yes No

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

Last modified November 05, 2020 at 4:28 PM PST: [Add usage for per-profile node affinity \(45da527a3\)](#)

[Edit this page](#) [Create child page](#) [Create an issue](#)

- [nodeSelector](#)
 - [Step Zero: Prerequisites](#)
 - [Step One: Attach label to the node](#)
 - [Step Two: Add a nodeSelector field to your pod configuration](#)
- [Interlude: built-in node labels](#)
- [Node isolation/restriction](#)
- [Affinity and anti-affinity](#)
 - [Node affinity](#)
 - [Inter-pod affinity and anti-affinity](#)
- [nodeName](#)
- [What's next](#)

Pod Overhead

FEATURE STATE: Kubernetes v1.18 [beta]

When you run a Pod on a Node, the Pod itself takes an amount of system resources. These resources are additional to the resources needed to run the container(s) inside the Pod. Pod Overhead is a feature for accounting for

the resources consumed by the Pod infrastructure on top of the container requests & limits.

In Kubernetes, the Pod's overhead is set at [admission](#) time according to the overhead associated with the Pod's [RuntimeClass](#).

When Pod Overhead is enabled, the overhead is considered in addition to the sum of container resource requests when scheduling a Pod. Similarly, Kubelet will include the Pod overhead when sizing the Pod cgroup, and when carrying out Pod eviction ranking.

Enabling Pod Overhead

You need to make sure that the PodOverhead [feature gate](#) is enabled (it is on by default as of 1.18) across your cluster, and a RuntimeClass is utilized which defines the overhead field.

Usage example

To use the PodOverhead feature, you need a RuntimeClass that defines the overhead field. As an example, you could use the following RuntimeClass definition with a virtualizing container runtime that uses around 120MiB per Pod for the virtual machine and the guest OS:

```
---
kind: RuntimeClass
apiVersion: node.k8s.io/v1
metadata:
  name: kata-fc
handler: kata-fc
overhead:
  podFixed:
    memory: "120Mi"
    cpu: "250m"
```

Workloads which are created which specify the kata-fc RuntimeClass handler will take the memory and cpu overheads into account for resource quota calculations, node scheduling, as well as Pod cgroup sizing.

Consider running the given example workload, test-pod:

```
apiVersion: v1
kind: Pod
metadata:
  name: test-pod
spec:
  runtimeClassName: kata-fc
  containers:
    - name: busybox-ctr
      image: busybox
      stdin: true
```

```
  tty: true
  resources:
    limits:
      cpu: 500m
      memory: 100Mi
  - name: nginx-ctr
    image: nginx
    resources:
      limits:
        cpu: 1500m
        memory: 100Mi
```

At admission time the `RuntimeClass` [admission controller](#) updates the workload's `PodSpec` to include the overhead as described in the `RuntimeClass`. If the `PodSpec` already has this field defined, the Pod will be rejected. In the given example, since only the `RuntimeClass` name is specified, the admission controller mutates the Pod to include an overhead.

After the `RuntimeClass` admission controller, you can check the updated `PodSpec`:

```
kubectl get pod test-pod -o jsonpath='{.spec.overhead}'
```

The output is:

```
map[cpu:250m memory:120Mi]
```

If a `ResourceQuota` is defined, the sum of container requests as well as the `overhead` field are counted.

When the kube-scheduler is deciding which node should run a new Pod, the scheduler considers that Pod's overhead as well as the sum of container requests for that Pod. For this example, the scheduler adds the requests and the overhead, then looks for a node that has 2.25 CPU and 320 MiB of memory available.

Once a Pod is scheduled to a node, the kubelet on that node creates a new [`cgroup`](#) for the Pod. It is within this pod that the underlying container runtime will create containers.

If the resource has a limit defined for each container (Guaranteed QoS or Burstable QoS with limits defined), the kubelet will set an upper limit for the pod `cgroup` associated with that resource (`cpu.cfs_quota_us` for CPU and `memory.limit_in_bytes` memory). This upper limit is based on the sum of the container limits plus the overhead defined in the `PodSpec`.

For CPU, if the Pod is Guaranteed or Burstable QoS, the kubelet will set `cpu.shares` based on the sum of container requests plus the overhead defined in the `PodSpec`.

Looking at our example, verify the container requests for the workload:

```
kubectl get pod test-pod -o jsonpath='{.spec.containers[*].resources.limits}'
```

The total container requests are 2000m CPU and 200MiB of memory:

```
map[cpu: 500m memory:100Mi] map[cpu:1500m memory:100Mi]
```

Check this against what is observed by the node:

```
kubectl describe node | grep test-pod -B2
```

The output shows 2250m CPU and 320MiB of memory are requested, which includes PodOverhead:

Namespace CPU Limits	Memory Requests	Name Memory Limits	AGE	CPU Requests
-----	-----	-----	---	-----
default 2250m (56%)	320Mi (1%)	test-pod 320Mi (1%)	36m	2250m (56%)

Verify Pod cgroup limits

Check the Pod's memory cgroups on the node where the workload is running. In the following example, [crictl](#) is used on the node, which provides a CLI for CRI-compatible container runtimes. This is an advanced example to show PodOverhead behavior, and it is not expected that users should need to check cgroups directly on the node.

First, on the particular node, determine the Pod identifier:

```
# Run this on the node where the Pod is scheduled  
POD_ID="$(sudo crictl pods --name test-pod -q)"
```

From this, you can determine the cgroup path for the Pod:

```
# Run this on the node where the Pod is scheduled  
sudo crictl inspectp -o=json $POD_ID | grep cgroupsPath
```

The resulting cgroup path includes the Pod's pause container. The Pod level cgroup is one directory above.

```
"cgroupsPath": "/kubepods/podd7f4b509-cf94-4951-9417-d1087c92a5b2/7ccf55aee35dd16aca4189c952d83487297f3cd760f1bbf09620e206e7d0c27a"
```

In this specific case, the pod cgroup path is `kubepods/podd7f4b509-cf94-4951-9417-d1087c92a5b2`. Verify the Pod level cgroup setting for memory:

```
# Run this on the node where the Pod is scheduled.  
# Also, change the name of the cgroup to match the cgroup  
allocated for your pod.
```

```
cat /sys/fs/cgroup/memory/kubepods/podd7f4b509-cf94-4951-9417-d1087c92a5b2/memory.limit_in_bytes
```

This is 320 MiB, as expected:

```
335544320
```

Observability

A `kube_pod_overhead` metric is available in [kube-state-metrics](#) to help identify when `PodOverhead` is being utilized and to help observe stability of workloads running with a defined Overhead. This functionality is not available in the 1.9 release of `kube-state-metrics`, but is expected in a following release. Users will need to build `kube-state-metrics` from source in the meantime.

What's next

- [RuntimeClass](#)
- [PodOverhead Design](#)

Feedback

Was this page helpful?

Yes No

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

Last modified October 23, 2020 at 8:57 PM PST: [RuntimeClass GA \(72a66b696\)](#)

[Edit this page](#) [Create child page](#) [Create an issue](#)

- [Enabling Pod Overhead](#)
- [Usage example](#)
- [Verify Pod cgroup limits](#)
 - [Observability](#)
- [What's next](#)

Resource Bin Packing for Extended Resources

FEATURE STATE: Kubernetes v1.16 [alpha]

The kube-scheduler can be configured to enable bin packing of resources along with extended resources using `RequestedToCapacityRatioResourceA`

llocation priority function. Priority functions can be used to fine-tune the kube-scheduler as per custom needs.

Enabling Bin Packing using RequestedToCapacityRatioResourceAllocation

Kubernetes allows the users to specify the resources along with weights for each resource to score nodes based on the request to capacity ratio. This allows users to bin pack extended resources by using appropriate parameters and improves the utilization of scarce resources in large clusters. The behavior of the RequestedToCapacityRatioResourceAllocation priority function can be controlled by a configuration option called requestedToCapacityRatioArguments. This argument consists of two parameters shape and resources. The shape parameter allows the user to tune the function as least requested or most requested based on utilization and score values. The resources parameter consists of name of the resource to be considered during scoring and weight specify the weight of each resource.

Below is an example configuration that sets requestedToCapacityRatioArguments to bin packing behavior for extended resources intel.com/foo and intel.com/bar.

```
apiVersion: v1
kind: Policy
# ...
priorities:
# ...
- name: RequestedToCapacityRatioPriority
  weight: 2
  argument:
    requestedToCapacityRatioArguments:
      shape:
        - utilization: 0
          score: 0
        - utilization: 100
          score: 10
      resources:
        - name: intel.com/foo
          weight: 3
        - name: intel.com/bar
          weight: 5
```

This feature is disabled by default

Tuning the Priority Function

shape is used to specify the behavior of the RequestedToCapacityRatioPriority function.

```
shape:
- utilization: 0
```

```
score: 0
- utilization: 100
  score: 10
```

The above arguments give the node a *score* of 0 if utilization is 0% and 10 for utilization 100%, thus enabling bin packing behavior. To enable least requested the *score* value must be reversed as follows.

```
shape:
- utilization: 0
  score: 100
- utilization: 100
  score: 0
```

resources is an optional parameter which defaults to:

```
resources:
- name: CPU
  weight: 1
- name: Memory
  weight: 1
```

It can be used to add extended resources as follows:

```
resources:
- name: intel.com/foo
  weight: 5
- name: CPU
  weight: 3
- name: Memory
  weight: 1
```

The *weight* parameter is optional and is set to 1 if not specified. Also, the *weight* cannot be set to a negative value.

Node scoring for capacity allocation

This section is intended for those who want to understand the internal details of this feature. Below is an example of how the node score is calculated for a given set of values.

Requested resources:

```
intel.com/foo : 2
Memory: 256MB
CPU: 2
```

Resource weights:

```
intel.com/foo : 5
Memory: 1
CPU: 3
```

```
FunctionShapePoint {{0, 0}, {100, 10}}
```

Node 1 spec:

```
Available:  
intel.com/foo: 4  
Memory: 1 GB  
CPU: 8
```

```
Used:  
intel.com/foo: 1  
Memory: 256MB  
CPU: 1
```

Node score:

```
intel.com/foo = resourceScoringFunction((2+1),4)  
= (100 - ((4-3)*100/4)  
= (100 - 25)  
= 75 # requested + used =  
75% * available  
= rawScoringFunction(75)  
= 7 # floor(75/10)  
  
Memory = resourceScoringFunction((256+256),1024)  
= (100 - ((1024-512)*100/1024))  
= 50 # requested + used =  
50% * available  
= rawScoringFunction(50)  
= 5 # floor(50/10)  
  
CPU = resourceScoringFunction((2+1),8)  
= (100 - ((8-3)*100/8))  
= 37.5 # requested + used =  
37.5% * available  
= rawScoringFunction(37.5)  
= 3 # floor(37.5/10)  
  
NodeScore = (7 * 5) + (5 * 1) + (3 * 3) / (5 + 1 + 3)  
= 5
```

Node 2 spec:

```
Available:  
intel.com/foo: 8  
Memory: 1GB  
CPU: 8
```

```
Used:  
intel.com/foo: 2  
Memory: 512MB  
CPU: 6
```

Node score:

```

intel.com/foo = resourceScoringFunction((2+2),8)
= (100 - ((8-4)*100/8)
= (100 - 50)
= 50
= rawScoringFunction(50)
= 5

Memory        = resourceScoringFunction((256+512),1024)
= (100 - ((1024-768)*100/1024))
= 75
= rawScoringFunction(75)
= 7

CPU           = resourceScoringFunction((2+6),8)
= (100 - ((8-8)*100/8))
= 100
= rawScoringFunction(100)
= 10

NodeScore     = (5 * 5) + (7 * 1) + (10 * 3) / (5 + 1 + 3)
= 7

```

What's next

- Read more about the [scheduling framework](#)
- Read more about [scheduler configuration](#)

Feedback

Was this page helpful?

Yes No

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

Last modified November 17, 2020 at 2:33 PM PST: [Tweak page style for resource-bin-packing \(2c38cb1bd\)](#)

[Edit this page](#) [Create child page](#) [Create an issue](#)

- [Enabling Bin Packing using RequestedToCapacityRatioResourceAllocation](#)
 - [Tuning the Priority Function](#)
 - [Node scoring for capacity allocation](#)
- [What's next](#)

Eviction Policy

This page is an overview of Kubernetes' policy for eviction.

Eviction Policy

The [kubelet](#) proactively monitors for and prevents total starvation of a compute resource. In those cases, the kubelet can reclaim the starved resource by failing one or more Pods. When the kubelet fails a Pod, it terminates all of its containers and transitions its PodPhase to Failed. If the evicted Pod is managed by a Deployment, the Deployment creates another Pod to be scheduled by Kubernetes.

What's next

- Learn how to [configure out of resource handling](#) with eviction signals and thresholds.

Feedback

Was this page helpful?

Yes No

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

Last modified September 29, 2020 at 8:37 PM PST: [fix eviction policy content type \(3fc4a3e8b\)](#)

[Edit this page](#) [Create child page](#) [Create an issue](#)

- [Eviction Policy](#)
- [What's next](#)

Scheduling Framework

FEATURE STATE: Kubernetes v1.15 [alpha]

The scheduling framework is a pluggable architecture for Kubernetes Scheduler that makes scheduler customizations easy. It adds a new set of "plugin" APIs to the existing scheduler. Plugins are compiled into the scheduler. The APIs allow most scheduling features to be implemented as plugins, while keeping the scheduling "core" simple and maintainable. Refer to the [design proposal of the scheduling framework](#) for more technical information on the design of the framework.

Framework workflow

The Scheduling Framework defines a few extension points. Scheduler plugins register to be invoked at one or more extension points. Some of these plugins can change the scheduling decisions and some are informational only.

Each attempt to schedule one Pod is split into two phases, the **scheduling cycle** and the **binding cycle**.

Scheduling Cycle & Binding Cycle

The scheduling cycle selects a node for the Pod, and the binding cycle applies that decision to the cluster. Together, a scheduling cycle and binding cycle are referred to as a "scheduling context".

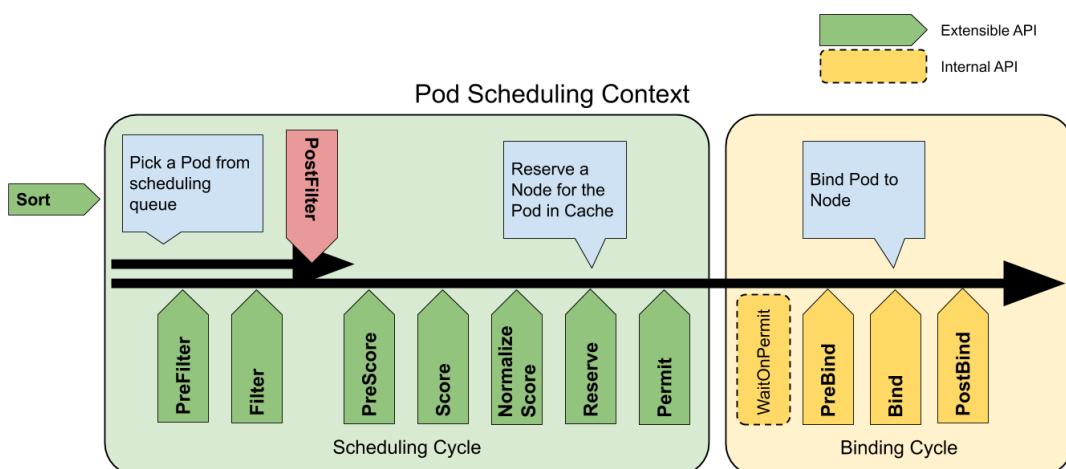
Scheduling cycles are run serially, while binding cycles may run concurrently.

A scheduling or binding cycle can be aborted if the Pod is determined to be unschedulable or if there is an internal error. The Pod will be returned to the queue and retried.

Extension points

The following picture shows the scheduling context of a Pod and the extension points that the scheduling framework exposes. In this picture "Filter" is equivalent to "Predicate" and "Scoring" is equivalent to "Priority function".

One plugin may register at multiple extension points to perform more complex or stateful tasks.



scheduling framework extension points

QueueSort

These plugins are used to sort Pods in the scheduling queue. A queue sort plugin essentially provides a `Less(Pod1, Pod2)` function. Only one queue sort plugin may be enabled at a time.

PreFilter

These plugins are used to pre-process info about the Pod, or to check certain conditions that the cluster or the Pod must meet. If a PreFilter plugin returns an error, the scheduling cycle is aborted.

Filter

These plugins are used to filter out nodes that cannot run the Pod. For each node, the scheduler will call filter plugins in their configured order. If any filter plugin marks the node as infeasible, the remaining plugins will not be called for that node. Nodes may be evaluated concurrently.

PostFilter

These plugins are called after Filter phase, but only when no feasible nodes were found for the pod. Plugins are called in their configured order. If any postFilter plugin marks the node as `Schedulable`, the remaining plugins will not be called. A typical PostFilter implementation is preemption, which tries to make the pod schedulable by preempting other Pods.

PreScore

These plugins are used to perform "pre-scoring" work, which generates a sharable state for Score plugins to use. If a PreScore plugin returns an error, the scheduling cycle is aborted.

Score

These plugins are used to rank nodes that have passed the filtering phase. The scheduler will call each scoring plugin for each node. There will be a well defined range of integers representing the minimum and maximum scores. After the [NormalizeScore](#) phase, the scheduler will combine node scores from all plugins according to the configured plugin weights.

NormalizeScore

These plugins are used to modify scores before the scheduler computes a final ranking of Nodes. A plugin that registers for this extension point will be called with the [Score](#) results from the same plugin. This is called once per plugin per scheduling cycle.

For example, suppose a plugin `BlinkingLightScorer` ranks Nodes based on how many blinking lights they have.

```
func ScoreNode(_ *v1.Pod, n *v1.Node) (int, error) {
    return getBlinkingLightCount(n)
}
```

However, the maximum count of blinking lights may be small compared to `NodeScoreMax`. To fix this, `BlinkingLightScorer` should also register for this extension point.

```
func NormalizeScores(scores map[string]int) {
    highest := 0
    for _, score := range scores {
        highest = max(highest, score)
    }
    for node, score := range scores {
        scores[node] = score*NodeScoreMax/highest
    }
}
```

If any `NormalizeScore` plugin returns an error, the scheduling cycle is aborted.

Note: Plugins wishing to perform "pre-reserve" work should use the `NormalizeScore` extension point.

Reserve

A plugin that implements the Reserve extension has two methods, namely `Reserve` and `Unreserve`, that back two informational scheduling phases called `Reserve` and `Unreserve`, respectively. Plugins which maintain runtime state (aka "stateful plugins") should use these phases to be notified by the scheduler when resources on a node are being reserved and unreserved for a given Pod.

The `Reserve` phase happens before the scheduler actually binds a Pod to its designated node. It exists to prevent race conditions while the scheduler waits for the bind to succeed. The `Reserve` method of each Reserve plugin may succeed or fail; if one `Reserve` method call fails, subsequent plugins are not executed and the `Reserve` phase is considered to have failed. If the `Reserve` method of all plugins succeed, the `Reserve` phase is considered to be successful and the rest of the scheduling cycle and the binding cycle are executed.

The `Unreserve` phase is triggered if the `Reserve` phase or a later phase fails. When this happens, the `Unreserve` method of **all** Reserve plugins will be executed in the reverse order of `Reserve` method calls. This phase exists to clean up the state associated with the reserved Pod.

Caution: The implementation of the `Unreserve` method in Reserve plugins must be idempotent and may not fail.

Permit

Permit plugins are invoked at the end of the scheduling cycle for each Pod, to prevent or delay the binding to the candidate node. A permit plugin can do one of the three things:

- 1. approve**

Once all Permit plugins approve a Pod, it is sent for binding.

- 2. deny**

If any Permit plugin denies a Pod, it is returned to the scheduling queue. This will trigger the Unreserve phase in [Reserve plugins](#).

- 3. wait (with a timeout)**

*If a Permit plugin returns "wait", then the Pod is kept in an internal "waiting" Pods list, and the binding cycle of this Pod starts but directly blocks until it gets approved. If a timeout occurs, **wait** becomes **deny** and the Pod is returned to the scheduling queue, triggering the Unreserve phase in [Reserve plugins](#).*

Note: While any plugin can access the list of "waiting" Pods and approve them (see [FrameworkHandle](#)), we expect only the permit plugins to approve binding of reserved Pods that are in "waiting" state. Once a Pod is approved, it is sent to the [PreBind](#) phase.

PreBind

These plugins are used to perform any work required before a Pod is bound. For example, a pre-bind plugin may provision a network volume and mount it on the target node before allowing the Pod to run there.

If any PreBind plugin returns an error, the Pod is [rejected](#) and returned to the scheduling queue.

Bind

*These plugins are used to bind a Pod to a Node. Bind plugins will not be called until all PreBind plugins have completed. Each bind plugin is called in the configured order. A bind plugin may choose whether or not to handle the given Pod. If a bind plugin chooses to handle a Pod, **the remaining bind plugins are skipped**.*

PostBind

This is an informational extension point. Post-bind plugins are called after a Pod is successfully bound. This is the end of a binding cycle, and can be used to clean up associated resources.

Plugin API

There are two steps to the plugin API. First, plugins must register and get configured, then they use the extension point interfaces. Extension point interfaces have the following form.

```
type Plugin interface {
    Name() string
}

type QueueSortPlugin interface {
    Plugin
    Less(*v1.Pod, *v1.Pod) bool
}

type PreFilterPlugin interface {
    Plugin
    PreFilter(context.Context, *framework.CycleState, *v1.Pod) error
}

// ...
```

Plugin configuration

You can enable or disable plugins in the scheduler configuration. If you are using Kubernetes v1.18 or later, most scheduling [plugins](#) are in use and enabled by default.

In addition to default plugins, you can also implement your own scheduling plugins and get them configured along with default plugins. You can visit [scheduler-plugins](#) for more details.

If you are using Kubernetes v1.18 or later, you can configure a set of plugins as a scheduler profile and then define multiple profiles to fit various kinds of workload. Learn more at [multiple profiles](#).

Feedback

Was this page helpful?

Yes No

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

Last modified October 14, 2020 at 7:04 PM PST: [Change outdated link](#) ([f566c22e6](#))

- [Scheduling Cycle & Binding Cycle](#)
- [Extension points](#)
 - [QueueSort](#)
 - [PreFilter](#)
 - [Filter](#)
 - [PostFilter](#)
 - [PreScore](#)
 - [Score](#)
 - [NormalizeScore](#)
 - [Reserve](#)
 - [Permit](#)
 - [PreBind](#)
 - [Bind](#)
 - [PostBind](#)
- [Plugin API](#)
- [Plugin configuration](#)

Scheduler Performance Tuning

FEATURE STATE: Kubernetes v1.14 [beta]

[kube-scheduler](#) is the Kubernetes default scheduler. It is responsible for placement of Pods on Nodes in a cluster.

Nodes in a cluster that meet the scheduling requirements of a Pod are called feasible Nodes for the Pod. The scheduler finds feasible Nodes for a Pod and then runs a set of functions to score the feasible Nodes, picking a Node with the highest score among the feasible ones to run the Pod. The scheduler then notifies the API server about this decision in a process called Binding.

This page explains performance tuning optimizations that are relevant for large Kubernetes clusters.

In large clusters, you can tune the scheduler's behaviour balancing scheduling outcomes between latency (new Pods are placed quickly) and accuracy (the scheduler rarely makes poor placement decisions).

You configure this tuning setting via `kube-scheduler` setting `percentageOfNodesToScore`. This `KubeSchedulerConfiguration` setting determines a threshold for scheduling nodes in your cluster.

Setting the threshold

The `percentageOfNodesToScore` option accepts whole numeric values between 0 and 100. The value 0 is a special number which indicates that the `kube-scheduler` should use its compiled-in default. If you set `percentageOfNodesToScore` above 100, `kube-scheduler` acts as if you had set a value of 100.

To change the value, edit the kube-scheduler configuration file (this is likely to be `/etc/kubernetes/config/kube-scheduler.yaml`), then restart the scheduler.

After you have made this change, you can run

```
kubectl get pods -n kube-system | grep kube-scheduler
```

to verify that the kube-scheduler component is healthy.

Node scoring threshold

To improve scheduling performance, the kube-scheduler can stop looking for feasible nodes once it has found enough of them. In large clusters, this saves time compared to a naive approach that would consider every node.

You specify a threshold for how many nodes are enough, as a whole number percentage of all the nodes in your cluster. The kube-scheduler converts this into an integer number of nodes. During scheduling, if the kube-scheduler has identified enough feasible nodes to exceed the configured percentage, the kube-scheduler stops searching for more feasible nodes and moves on to the [scoring phase](#).

[How the scheduler iterates over Nodes](#) describes the process in detail.

Default threshold

If you don't specify a threshold, Kubernetes calculates a figure using a linear formula that yields 50% for a 100-node cluster and yields 10% for a 5000-node cluster. The lower bound for the automatic value is 5%.

This means that, the kube-scheduler always scores at least 5% of your cluster no matter how large the cluster is, unless you have explicitly set `percentageOfNodesToScore` to be smaller than 5.

If you want the scheduler to score all nodes in your cluster, set `percentageOfNodesToScore` to 100.

Example

Below is an example configuration that sets `percentageOfNodesToScore` to 50%.

```
apiVersion: kubescheduler.config.k8s.io/v1alpha1
kind: KubeSchedulerConfiguration
algorithmSource:
  provider: DefaultProvider
...
percentageOfNodesToScore: 50
```

Tuning `percentageOfNodesToScore`

`percentageOfNodesToScore` must be a value between 1 and 100 with the default value being calculated based on the cluster size. There is also a hardcoded minimum value of 50 nodes.

Note:

In clusters with less than 50 feasible nodes, the scheduler still checks all the nodes, simply because there are not enough feasible nodes to stop the scheduler's search early.

In a small cluster, if you set a low value for `percentageOfNodesToScore`, your change will have no or little effect, for a similar reason.

If your cluster has several hundred Nodes or fewer, leave this configuration option at its default value. Making changes is unlikely to improve the scheduler's performance significantly.

An important detail to consider when setting this value is that when a smaller number of nodes in a cluster are checked for feasibility, some nodes are not sent to be scored for a given Pod. As a result, a Node which could possibly score a higher value for running the given Pod might not even be passed to the scoring phase. This would result in a less than ideal placement of the Pod.

You should avoid setting `percentageOfNodesToScore` very low so that kube-scheduler does not make frequent, poor Pod placement decisions. Avoid setting the percentage to anything below 10%, unless the scheduler's throughput is critical for your application and the score of nodes is not important. In other words, you prefer to run the Pod on any Node as long as it is feasible.

How the scheduler iterates over Nodes

This section is intended for those who want to understand the internal details of this feature.

In order to give all the Nodes in a cluster a fair chance of being considered for running Pods, the scheduler iterates over the nodes in a round robin fashion. You can imagine that Nodes are in an array. The scheduler starts from the start of the array and checks feasibility of the nodes until it finds enough Nodes as specified by `percentageOfNodesToScore`. For the next Pod, the scheduler continues from the point in the Node array that it stopped at when checking feasibility of Nodes for the previous Pod.

If Nodes are in multiple zones, the scheduler iterates over Nodes in various zones to ensure that Nodes from different zones are considered in the feasibility checks. As an example, consider six nodes in two zones:

`Zone 1: Node 1, Node 2, Node 3, Node 4`

`Zone 2: Node 5, Node 6`

The Scheduler evaluates feasibility of the nodes in this order:

`Node 1, Node 5, Node 2, Node 6, Node 3, Node 4`

After going over all the Nodes, it goes back to Node 1.

Feedback

Was this page helpful?

Yes No

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

Last modified September 14, 2020 at 11:13 AM PST: [Remove reference to componentstatus \(e566a726d\)](#)

[Edit this page](#) [Create child page](#) [Create an issue](#)

- ◦ [Setting the threshold](#)
- [Node scoring threshold](#)
 - [Default threshold](#)
- [Example](#)
- [Tuning percentageOfNodesToScore](#)
- [How the scheduler iterates over Nodes](#)

Cluster Administration

Lower-level detail relevant to creating or administering a Kubernetes cluster.

The cluster administration overview is for anyone creating or administering a Kubernetes cluster. It assumes some familiarity with core Kubernetes [concepts](#).

Planning a cluster

See the guides in [Setup](#) for examples of how to plan, set up, and configure Kubernetes clusters. The solutions listed in this article are called distros.

Note: Not all distros are actively maintained. Choose distros which have been tested with a recent version of Kubernetes.

Before choosing a guide, here are some considerations:

- Do you just want to try out Kubernetes on your computer, or do you want to build a high-availability, multi-node cluster? Choose distros best suited for your needs.
- Will you be using a **hosted Kubernetes cluster**, such as [Google Kubernetes Engine](#), or **hosting your own cluster**?
- Will your cluster be **on-premises**, or **in the cloud (IaaS)**? Kubernetes does not directly support hybrid clusters. Instead, you can set up multiple clusters.
- If you are **configuring Kubernetes on-premises**, consider which [networking model](#) fits best.
- Will you be running Kubernetes on "**bare metal**" **hardware** or on **virtual machines (VMs)**?
- Do you **just want to run a cluster**, or do you expect to do **active development of Kubernetes project code**? If the latter, choose an actively-developed distro. Some distros only use binary releases, but offer a greater variety of choices.
- Familiarize yourself with the [components](#) needed to run a cluster.

Managing a cluster

- Learn how to [manage nodes](#).
- Learn how to set up and manage the [resource quota](#) for shared clusters.

Securing a cluster

- [Certificates](#) describes the steps to generate certificates using different tool chains.
- [Kubernetes Container Environment](#) describes the environment for Kubelet managed containers on a Kubernetes node.
- [Controlling Access to the Kubernetes API](#) describes how Kubernetes implements access control for its own API.
- [Authenticating](#) explains authentication in Kubernetes, including the various authentication options.
- [Authorization](#) is separate from authentication, and controls how HTTP calls are handled.
- [Using Admission Controllers](#) explains plug-ins which intercepts requests to the Kubernetes API server after authentication and authorization.
- [Using Sysctls in a Kubernetes Cluster](#) describes to an administrator how to use the `sysctl` command-line tool to set kernel parameters .
- [Auditing](#) describes how to interact with Kubernetes' audit logs.

Securing the kubelet

- [Control Plane-Node communication](#)
- [TLS bootstrapping](#)
- [Kubelet authentication/authorization](#)

Optional Cluster Services

- [DNS Integration](#) describes how to resolve a DNS name directly to a Kubernetes service.
- [Logging and Monitoring Cluster Activity](#) explains how logging in Kubernetes works and how to implement it.

Certificates

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

easyrsa

easyrsa can manually generate certificates for your cluster.

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

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

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

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

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

```
./easyrsa --subject-alt-name="IP:${MASTER_IP}, \"\n"IP:${MASTER_CLUSTER_IP}, \"\n"DNS:kubernetes, \"\n
```

```
"DNS:kubernetes.default,"\  
"DNS:kubernetes.default.svc,"\  
"DNS:kubernetes.default.svc.cluster,"\  
"DNS:kubernetes.default.svc.cluster.local" \  
--days=10000 \  
build-server-full server nopass
```

4. Copy `pki/ca.crt`, `pki/issued/server.crt`, and `pki/private/server.key` to your directory.
5. Fill in and add the following parameters into the API server start parameters:

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

openssl

openssl can manually generate certificates for your cluster.

1. Generate a `ca.key` with 2048bit:

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

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

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

3. Generate a `server.key` with 2048bit:

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

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

```
[ req ]  
default_bits = 2048  
prompt = no  
default_md = sha256  
req_extensions = req_ext  
distinguished_name = dn  
  
[ dn ]  
C = <country>  
ST = <state>
```

```

L = <city>
O = <organization>
OU = <organization unit>
CN = <MASTER_IP>

[ req_ext ]
subjectAltName = @alt_names

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

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

```

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

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

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

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

7. View the certificate:

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

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

cfssl

cfssl is another tool for certificate generation.

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

```
curl -L https://github.com/cloudflare/cfssl/releases/
download/v1.4.1/cfssl_1.4.1_linux_amd64 -o cfssl
chmod +x cfssl
curl -L https://github.com/cloudflare/cfssl/releases/
```

```
download/v1.4.1/cfssljson_1.4.1_linux_amd64 -o cfssljson  
chmod +x cfssljson  
curl -L https://github.com/cloudflare/cfssl/releases/  
download/v1.4.1/cfssl-certinfo_1.4.1_linux_amd64 -o 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 [here](#).

Feedback

Was this page helpful?

Yes *No*

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

*Last modified May 30, 2020 at 3:10 PM PST: [add en pages \(ecc27bbbe\)](#)
[Edit this page](#) [Create child page](#) [Create an issue](#)*

- - [easyrsa](#)
 - [openssl](#)
 - [cfssl](#)
- [Distributing Self-Signed CA Certificate](#)
- [Certificates API](#)

Managing Resources

You've deployed your application and exposed it via a service. Now what? Kubernetes provides a number of tools to help you manage your application deployment, including scaling and updating. Among the features that we will discuss in more depth are [configuration files](#) and [labels](#).

Organizing resource configurations

Many applications require multiple resources to be created, such as a Deployment and a Service. Management of multiple resources can be simplified by grouping them together in the same file (separated by `---` in YAML). For example:



```
apiVersion: v1
kind: Service
metadata:
  name: my-nginx-svc
  labels:
    app: nginx
spec:
  type: LoadBalancer
  ports:
  - port: 80
  selector:
    app: nginx
```

```
---  
apiVersion: apps/v1
kind: Deployment
metadata:
  name: my-nginx
  labels:
    app: nginx
spec:
  replicas: 3
  selector:
    matchLabels:
      app: nginx
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
      - name: nginx
        image: nginx:1.14.2
        ports:
        - containerPort: 80
```

Multiple resources can be created the same way as a single resource:

```
kubectl apply -f https://k8s.io/examples/application/nginx-
app.yaml
```

```
service/my-nginx-svc created
deployment.apps/my-nginx created
```

The resources will be created in the order they appear in the file. Therefore, it's best to specify the service first, since that will ensure the scheduler can spread the pods associated with the service as they are created by the controller(s), such as Deployment.

kubectl apply also accepts multiple -f arguments:

```
kubectl apply -f https://k8s.io/examples/application/nginx/nginx-svc.yaml -f https://k8s.io/examples/application/nginx/nginx-deployment.yaml
```

And a directory can be specified rather than or in addition to individual files:

```
kubectl apply -f https://k8s.io/examples/application/nginx/
```

kubectl will read any files with suffixes .yaml, .yml, or .json.

It is a recommended practice to put resources related to the same microservice or application tier into the same file, and to group all of the files associated with your application in the same directory. If the tiers of your application bind to each other using DNS, then you can then simply deploy all of the components of your stack en masse.

A URL can also be specified as a configuration source, which is handy for deploying directly from configuration files checked into github:

```
kubectl apply -f https://raw.githubusercontent.com/kubernetes/website/master/content/en/examples/application/nginx/nginx-deployment.yaml
```

```
deployment.apps/my-nginx created
```

Bulk operations in kubectl

Resource creation isn't the only operation that kubectl can perform in bulk. It can also extract resource names from configuration files in order to perform other operations, in particular to delete the same resources you created:

```
kubectl delete -f https://k8s.io/examples/application/nginx-app.yaml
```

```
deployment.apps "my-nginx" deleted  
service "my-nginx-svc" deleted
```

In the case of just two resources, it's also easy to specify both on the command line using the resource/name syntax:

```
kubectl delete deployments/my-nginx services/my-nginx-svc
```

For larger numbers of resources, you'll find it easier to specify the selector (label query) specified using -l or --selector, to filter resources by their labels:

```
kubectl delete deployment,services -l app=nginx
```

```
deployment.apps "my-nginx" deleted  
service "my-nginx-svc" deleted
```

Because `kubectl` outputs resource names in the same syntax it accepts, it's easy to chain operations using `$()` or `xargs`:

```
kubectl get $(kubectl create -f docs/concepts/cluster-administration/nginx/ -o name | grep service)
```

NAME	TYPE	CLUSTER-IP	EXTERNAL-IP	PORT(S)
my-nginx-svc	LoadBalancer	10.0.0.208	<pending>	80/TCP

With the above commands, we first create resources under `examples/application/nginx/` and print the resources created with `-o name` output format (print each resource as resource/name). Then we `grep` only the "service", and then print it with `kubectl get`.

If you happen to organize your resources across several subdirectories within a particular directory, you can recursively perform the operations on the subdirectories also, by specifying `--recursive` or `-R` alongside the `--filename`, `-f` flag.

For instance, assume there is a directory `project/k8s/development` that holds all of the [manifests](#) needed for the development environment, organized by resource type:

```
project/k8s/development
  configmap
    my-configmap.yaml
  deployment
    my-deployment.yaml
  pvc
    my-pvc.yaml
```

By default, performing a bulk operation on `project/k8s/development` will stop at the first level of the directory, not processing any subdirectories. If we had tried to create the resources in this directory using the following command, we would have encountered an error:

```
kubectl apply -f project/k8s/development
```

```
error: you must provide one or more resources by argument or
filename (.json|.yaml|.yml|stdin)
```

Instead, specify the `--recursive` or `-R` flag with the `--filename`, `-f` flag as such:

```
kubectl apply -f project/k8s/development --recursive
```

```
configmap/my-config created
deployment.apps/my-deployment created
persistentvolumeclaim/my-pvc created
```

The `--recursive` flag works with any operation that accepts the `--filename`, `-f` flag such as: `kubectl {create, get, delete, describe, rollout}` etc.

The `--recursive` flag also works when multiple `-f` arguments are provided:

```
kubectl apply -f project/k8s/namespaces -f project/k8s/development --recursive
```

```
namespace/development created
namespace/staging created
configmap/my-config created
deployment.apps/my-deployment created
persistentvolumeclaim/my-pvc created
```

If you're interested in learning more about `kubectl`, go ahead and read [kubectl Overview](#).

Using labels effectively

The examples we've used so far apply at most a single label to any resource. There are many scenarios where multiple labels should be used to distinguish sets from one another.

For instance, different applications would use different values for the `app` label, but a multi-tier application, such as the [guestbook example](#), would additionally need to distinguish each tier. The frontend could carry the following labels:

```
labels:
  app: guestbook
  tier: frontend
```

while the Redis master and slave would have different `tier` labels, and perhaps even an additional `role` label:

```
labels:
  app: guestbook
  tier: backend
  role: master
```

and

```
labels:
  app: guestbook
  tier: backend
  role: slave
```

The labels allow us to slice and dice our resources along any dimension specified by a label:

```
kubectl apply -f examples/guestbook/all-in-one/guestbook-all-in-one.yaml
kubectl get pods -Lapp -Ltier -Lrole
```

NAME	READY	STATUS	RESTARTS
AGE	APP	TIER	ROLE
guestbook-fe-4nlpb 1m	guestbook	frontend	1/1 <none>
guestbook-fe-ght6d 1m	guestbook	frontend	1/1 <none>
guestbook-fe-jpy62 1m	guestbook	frontend	1/1 <none>
guestbook-redis-master-5pg3b 1m	guestbook	backend	1/1 master
guestbook-redis-slave-2q2yf 1m	guestbook	backend	1/1 slave
guestbook-redis-slave-qgazl 1m	guestbook	backend	1/1 slave
my-nginx-divi2 29m	nginx	<none>	1/1 <none>
my-nginx-o0ef1 29m	nginx	<none>	1/1 <none>

```
kubectl get pods -lapp=guestbook,role=slave
```

NAME	READY	STATUS	RESTARTS	AGE
guestbook-redis-slave-2q2yf	1/1	Running	0	3m
guestbook-redis-slave-qgazl	1/1	Running	0	3m

Canary deployments

Another scenario where multiple labels are needed is to distinguish deployments of different releases or configurations of the same component. It is common practice to deploy a canary of a new application release (specified via image tag in the pod template) side by side with the previous release so that the new release can receive live production traffic before fully rolling it out.

For instance, you can use a `track` label to differentiate different releases.

The primary, stable release would have a `track` label with value as `stable`:

```
name: frontend
replicas: 3
...
labels:
  app: guestbook
  tier: frontend
  track: stable
...
image: gb-frontend:v3
```

and then you can create a new release of the guestbook frontend that carries the `track` label with different value (i.e. `canary`), so that two sets of pods would not overlap:

```
name: frontend-canary
replicas: 1
...
labels:
  app: guestbook
  tier: frontend
  track: canary
...
image: gb-frontend:v4
```

The frontend service would span both sets of replicas by selecting the common subset of their labels (i.e. omitting the `track` label), so that the traffic will be redirected to both applications:

```
selector:
  app: guestbook
  tier: frontend
```

You can tweak the number of replicas of the stable and canary releases to determine the ratio of each release that will receive live production traffic (in this case, 3:1). Once you're confident, you can update the stable track to the new application release and remove the canary one.

For a more concrete example, check the [tutorial of deploying Ghost](#).

Updating labels

Sometimes existing pods and other resources need to be relabeled before creating new resources. This can be done with `kubectl label`. For example, if you want to label all your nginx pods as frontend tier, simply run:

```
kubectl label pods -l app=nginx tier=fe
```

```
pod/my-nginx-2035384211-j5fhi labeled
pod/my-nginx-2035384211-u2c7e labeled
pod/my-nginx-2035384211-u3t6x labeled
```

This first filters all pods with the label "app=nginx", and then labels them with the "tier=fe". To see the pods you just labeled, run:

```
kubectl get pods -l app=nginx -L tier
```

NAME	READY	STATUS	RESTARTS
AGE	TIER		
my-nginx-2035384211-j5fhi	1/1	Running	0
23m fe			
my-nginx-2035384211-u2c7e	1/1	Running	0
23m fe			

my-nginx-2035384211-u3t6x	1/1	Running	0
23m	fe		

This outputs all "app=nginx" pods, with an additional label column of pods' tier (specified with `-L` or `--label-columns`).

For more information, please see [labels](#) and [kubectl label](#).

Updating annotations

Sometimes you would want to attach annotations to resources. Annotations are arbitrary non-identifying metadata for retrieval by API clients such as tools, libraries, etc. This can be done with `kubectl annotate`. For example:

```
kubectl annotate pods my-nginx-v4-9gw19 description='my frontend  
running nginx'  
kubectl get pods my-nginx-v4-9gw19 -o yaml
```

```
apiVersion: v1  
kind: pod  
metadata:  
  annotations:  
    description: my frontend running nginx  
...
```

For more information, please see [annotations](#) and [kubectl annotate](#) document.

Scaling your application

When load on your application grows or shrinks, it's easy to scale with `kubectl`. For instance, to decrease the number of nginx replicas from 3 to 1, do:

```
kubectl scale deployment/my-nginx --replicas=1
```

```
deployment.apps/my-nginx scaled
```

Now you only have one pod managed by the deployment.

```
kubectl get pods -l app=nginx
```

NAME	READY	STATUS	RESTARTS	AGE
my-nginx-2035384211-j5fhi	1/1	Running	0	30m

To have the system automatically choose the number of nginx replicas as needed, ranging from 1 to 3, do:

```
kubectl autoscale deployment/my-nginx --min=1 --max=3
```

```
horizontalpodautoscaler.autoscaling/my-nginx autoscaled
```

Now your nginx replicas will be scaled up and down as needed, automatically.

For more information, please see [kubectl scale](#), [kubectl autoscale](#) and [horizontal pod autoscaler](#) document.

In-place updates of resources

Sometimes it's necessary to make narrow, non-disruptive updates to resources you've created.

kubectl apply

It is suggested to maintain a set of configuration files in source control (see [configuration as code](#)), so that they can be maintained and versioned along with the code for the resources they configure. Then, you can use [kubectl apply](#) to push your configuration changes to the cluster.

This command will compare the version of the configuration that you're pushing with the previous version and apply the changes you've made, without overwriting any automated changes to properties you haven't specified.

```
kubectl apply -f https://k8s.io/examples/application/nginx/nginx-deployment.yaml  
deployment.apps/my-nginx configured
```

Note that kubectl apply attaches an annotation to the resource in order to determine the changes to the configuration since the previous invocation. When it's invoked, kubectl apply does a three-way diff between the previous configuration, the provided input and the current configuration of the resource, in order to determine how to modify the resource.

Currently, resources are created without this annotation, so the first invocation of kubectl apply will fall back to a two-way diff between the provided input and the current configuration of the resource. During this first invocation, it cannot detect the deletion of properties set when the resource was created. For this reason, it will not remove them.

All subsequent calls to kubectl apply, and other commands that modify the configuration, such as kubectl replace and kubectl edit, will update the annotation, allowing subsequent calls to kubectl apply to detect and perform deletions using a three-way diff.

kubectl edit

Alternatively, you may also update resources with kubectl edit:

```
kubectl edit deployment/my-nginx
```

This is equivalent to first get the resource, edit it in text editor, and then apply the resource with the updated version:

```
kubectl get deployment my-nginx -o yaml > /tmp/nginx.yaml  
vi /tmp/nginx.yaml  
# do some edit, and then save the file  
  
kubectl apply -f /tmp/nginx.yaml  
deployment.apps/my-nginx configured  
  
rm /tmp/nginx.yaml
```

This allows you to do more significant changes more easily. Note that you can specify the editor with your `EDITOR` or `KUBE_EDITOR` environment variables.

For more information, please see [kubectl edit](#) document.

kubectl patch

You can use `kubectl patch` to update API objects in place. This command supports JSON patch, JSON merge patch, and strategic merge patch. See [Update API Objects in Place Using kubectl patch](#) and [kubectl patch](#).

Disruptive updates

In some cases, you may need to update resource fields that cannot be updated once initialized, or you may just want to make a recursive change immediately, such as to fix broken pods created by a Deployment. To change such fields, use `replace --force`, which deletes and re-creates the resource. In this case, you can simply modify your original configuration file:

```
kubectl replace -f https://k8s.io/examples/application/nginx/  
nginx-deployment.yaml --force
```

```
deployment.apps/my-nginx deleted  
deployment.apps/my-nginx replaced
```

Updating your application without a service outage

At some point, you'll eventually need to update your deployed application, typically by specifying a new image or image tag, as in the canary deployment scenario above. `kubectl` supports several update operations, each of which is applicable to different scenarios.

We'll guide you through how to create and update applications with Deployments.

Let's say you were running version 1.14.2 of nginx:

```
kubectl create deployment my-nginx --image=nginx:1.14.2  
deployment.apps/my-nginx created
```

with 3 replicas (so the old and new revisions can coexist):

```
kubectl scale deployment my-nginx --current-replicas=1 --replicas=3
```

```
deployment.apps/my-nginx scaled
```

To update to version 1.16.1, simply change `.spec.template.spec.containers[0].image` from `nginx:1.14.2` to `nginx:1.16.1`, with the `kubectl` commands we learned above.

```
kubectl edit deployment/my-nginx
```

That's it! The Deployment will declaratively update the deployed nginx application progressively behind the scene. It ensures that only a certain number of old replicas may be down while they are being updated, and only a certain number of new replicas may be created above the desired number of pods. To learn more details about it, visit [Deployment page](#).

What's next

- Learn about [how to use kubectl for application introspection and debugging](#).
- See [Configuration Best Practices and Tips](#).

Feedback

Was this page helpful?

Yes No

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

Last modified July 17, 2020 at 4:11 PM PST: [Fix links in concepts section \(2\) \(c8f470487\)](#)

[Edit this page](#) [Create child page](#) [Create an issue](#)

- [Organizing resource configurations](#)
- [Bulk operations in kubectl](#)
- [Using labels effectively](#)
- [Canary deployments](#)
- [Updating labels](#)
- [Updating annotations](#)
- [Scaling your application](#)
- [In-place updates of resources](#)
 - [kubectl apply](#)
 - [kubectl edit](#)
 - [kubectl patch](#)
- [Disruptive updates](#)
- [Updating your application without a service outage](#)

- [What's next](#)

Cluster Networking

Networking is a central part of Kubernetes, but it can be challenging to understand exactly how it is expected to work. There are 4 distinct networking problems to address:

1. *Highly-coupled container-to-container communications: this is solved by [Pods](#) and [localhost](#) communications.*
2. *Pod-to-Pod communications: this is the primary focus of this document.*
3. *Pod-to-Service communications: this is covered by [services](#).*
4. *External-to-Service communications: this is covered by [services](#).*

Kubernetes is all about sharing machines between applications. Typically, sharing machines requires ensuring that two applications do not try to use the same ports. Coordinating ports across multiple developers is very difficult to do at scale and exposes users to cluster-level issues outside of their control.

Dynamic port allocation brings a lot of complications to the system - every application has to take ports as flags, the API servers have to know how to insert dynamic port numbers into configuration blocks, services have to know how to find each other, etc. Rather than deal with this, Kubernetes takes a different approach.

The Kubernetes network model

Every Pod gets its own IP address. This means you do not need to explicitly create links between Pods and you almost never need to deal with mapping container ports to host ports. This creates a clean, backwards-compatible model where Pods can be treated much like VMs or physical hosts from the perspectives of port allocation, naming, service discovery, load balancing, application configuration, and migration.

Kubernetes imposes the following fundamental requirements on any networking implementation (barring any intentional network segmentation policies):

- *pods on a node can communicate with all pods on all nodes without NAT*
- *agents on a node (e.g. system daemons, kubelet) can communicate with all pods on that node*

Note: For those platforms that support Pods running in the host network (e.g. Linux):

- *pods in the host network of a node can communicate with all pods on all nodes without NAT*

This model is not only less complex overall, but it is principally compatible with the desire for Kubernetes to enable low-friction porting of apps from VMs to containers. If your job previously ran in a VM, your VM had an IP and could talk to other VMs in your project. This is the same basic model.

Kubernetes IP addresses exist at the Pod scope - containers within a Pod share their network namespaces - including their IP address and MAC address. This means that containers within a Pod can all reach each other's ports on localhost. This also means that containers within a Pod must coordinate port usage, but this is no different from processes in a VM. This is called the "IP-per-pod" model.

How this is implemented is a detail of the particular container runtime in use.

It is possible to request ports on the Node itself which forward to your Pod (called host ports), but this is a very niche operation. How that forwarding is implemented is also a detail of the container runtime. The Pod itself is blind to the existence or non-existence of host ports.

How to implement the Kubernetes networking model

There are a number of ways that this network model can be implemented. This document is not an exhaustive study of the various methods, but hopefully serves as an introduction to various technologies and serves as a jumping-off point.

The following networking options are sorted alphabetically - the order does not imply any preferential status.

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

ACI

[Cisco Application Centric Infrastructure](#) offers an integrated overlay and underlay SDN solution that supports containers, virtual machines, and bare metal servers. [ACI](#) provides container networking integration for ACI. An overview of the integration is provided [here](#).

Antrea

Project [Antrea](#) is an open-source Kubernetes networking solution intended to be Kubernetes native. It leverages Open vSwitch as the networking data plane. Open vSwitch is a high-performance programmable virtual switch that supports both Linux and Windows. Open vSwitch enables Antrea to

implement Kubernetes Network Policies in a high-performance and efficient manner. Thanks to the "programmable" characteristic of Open vSwitch, Antrea is able to implement an extensive set of networking and security features and services on top of Open vSwitch.

AOS from Apstra

[AOS](#) is an Intent-Based Networking system that creates and manages complex datacenter environments from a simple integrated platform. AOS leverages a highly scalable distributed design to eliminate network outages while minimizing costs.

The AOS Reference Design currently supports Layer-3 connected hosts that eliminate legacy Layer-2 switching problems. These Layer-3 hosts can be Linux servers (Debian, Ubuntu, CentOS) that create BGP neighbor relationships directly with the top of rack switches (TORs). AOS automates the routing adjacencies and then provides fine grained control over the route health injections (RHI) that are common in a Kubernetes deployment.

AOS has a rich set of REST API endpoints that enable Kubernetes to quickly change the network policy based on application requirements. Further enhancements will integrate the AOS Graph model used for the network design with the workload provisioning, enabling an end to end management system for both private and public clouds.

AOS supports the use of common vendor equipment from manufacturers including Cisco, Arista, Dell, Mellanox, HPE, and a large number of white-box systems and open network operating systems like Microsoft SONiC, Dell OPX, and Cumulus Linux.

Details on how the AOS system works can be accessed here: <https://www.apstra.com/products/how-it-works/>

AWS VPC CNI for Kubernetes

The [AWS VPC CNI](#) offers integrated AWS Virtual Private Cloud (VPC) networking for Kubernetes clusters. This CNI plugin offers high throughput and availability, low latency, and minimal network jitter. Additionally, users can apply existing AWS VPC networking and security best practices for building Kubernetes clusters. This includes the ability to use VPC flow logs, VPC routing policies, and security groups for network traffic isolation.

Using this CNI plugin allows Kubernetes pods to have the same IP address inside the pod as they do on the VPC network. The CNI allocates AWS Elastic Networking Interfaces (ENIs) to each Kubernetes node and using the secondary IP range from each ENI for pods on the node. The CNI includes controls for pre-allocation of ENIs and IP addresses for fast pod startup times and enables large clusters of up to 2,000 nodes.

Additionally, the CNI can be run alongside [Calico for network policy enforcement](#). The AWS VPC CNI project is open source with [documentation on GitHub](#).

Azure CNI for Kubernetes

Azure CNI is an [open source](#) plugin that integrates Kubernetes Pods with an Azure Virtual Network (also known as VNet) providing network performance at par with VMs. Pods can connect to peered VNet and to on-premises over Express Route or site-to-site VPN and are also directly reachable from these networks. Pods can access Azure services, such as storage and SQL, that are protected by Service Endpoints or Private Link. You can use VNet security policies and routing to filter Pod traffic. The plugin assigns VNet IPs to Pods by utilizing a pool of secondary IPs pre-configured on the Network Interface of a Kubernetes node.

Azure CNI is available natively in the [Azure Kubernetes Service \(AKS\)](#).

Big Cloud Fabric from Big Switch Networks

Big Cloud Fabric is a cloud native networking architecture, designed to run Kubernetes in private cloud/on-premises environments. Using unified physical & virtual SDN, Big Cloud Fabric tackles inherent container networking problems such as load balancing, visibility, troubleshooting, security policies & container traffic monitoring.

With the help of the Big Cloud Fabric's virtual pod multi-tenant architecture, container orchestration systems such as Kubernetes, RedHat OpenShift, Mesosphere DC/OS & Docker Swarm will be natively integrated alongside with VM orchestration systems such as VMware, OpenStack & Nutanix. Customers will be able to securely inter-connect any number of these clusters and enable inter-tenant communication between them if needed.

BCF was recognized by Gartner as a visionary in the latest [Magic Quadrant](#). One of the BCF Kubernetes on-premises deployments (which includes Kubernetes, DC/OS & VMware running on multiple DCs across different geographic regions) is also referenced [here](#).

Calico

Calico is an open source networking and network security solution for containers, virtual machines, and native host-based workloads. Calico supports multiple data planes including: a pure Linux eBPF dataplane, a standard Linux networking dataplane, and a Windows HNS dataplane. Calico provides a full networking stack but can also be used in conjunction with [cloud provider CNIs](#) to provide network policy enforcement.

Cilium

Cilium is open source software for providing and transparently securing network connectivity between application containers. Cilium is L7/HTTP aware and can enforce network policies on L3-L7 using an identity based security model that is decoupled from network addressing, and it can be used in combination with other CNI plugins.

CNI-Genie from Huawei

[CNI-Genie](#) is a CNI plugin that enables Kubernetes to [simultaneously have access to different implementations](#) of the [Kubernetes network model](#) in runtime. This includes any implementation that runs as a [CNI plugin](#), such as [Flannel](#), [Calico](#), [Romana](#), [Weave-net](#).

CNI-Genie also supports [assigning multiple IP addresses to a pod](#), each from a different CNI plugin.

cni-ipvlan-vpc-k8s

[cni-ipvlan-vpc-k8s](#) contains a set of CNI and IPAM plugins to provide a simple, host-local, low latency, high throughput, and compliant networking stack for Kubernetes within Amazon Virtual Private Cloud (VPC) environments by making use of Amazon Elastic Network Interfaces (ENI) and binding AWS-managed IPs into Pods using the Linux kernel's IPvlan driver in L2 mode.

The plugins are designed to be straightforward to configure and deploy within a VPC. Kubelets boot and then self-configure and scale their IP usage as needed without requiring the often recommended complexities of administering overlay networks, BGP, disabling source/destination checks, or adjusting VPC route tables to provide per-instance subnets to each host (which is limited to 50-100 entries per VPC). In short, cni-ipvlan-vpc-k8s significantly reduces the network complexity required to deploy Kubernetes at scale within AWS.

Coil

[Coil](#) is a CNI plugin designed for ease of integration, providing flexible egress networking. Coil operates with a low overhead compared to bare metal, and allows you to define arbitrary egress NAT gateways for external networks.

Contiv

[Contiv](#) provides configurable networking (native l3 using BGP, overlay using vxlan, classic l2, or Cisco-SDN/ACI) for various use cases. [Contiv](#) is all open sourced.

Contrail / Tungsten Fabric

[Contrail](#), based on [Tungsten Fabric](#), is a truly open, multi-cloud network virtualization and policy management platform. Contrail and Tungsten Fabric are integrated with various orchestration systems such as Kubernetes, OpenShift, OpenStack and Mesos, and provide different isolation modes for virtual machines, containers/pods and bare metal workloads.

DANM

[DANM](#) is a networking solution for telco workloads running in a Kubernetes cluster. It's built up from the following components:

- A CNI plugin capable of provisioning IPVLAN interfaces with advanced features
- An in-built IPAM module with the capability of managing multiple, cluster-wide, discontinuous L3 networks and provide a dynamic, static, or no IP allocation scheme on-demand
- A CNI metaplugin capable of attaching multiple network interfaces to a container, either through its own CNI, or through delegating the job to any of the popular CNI solution like SRI-OV, or Flannel in parallel
- A Kubernetes controller capable of centrally managing both VxLAN and VLAN interfaces of all Kubernetes hosts
- Another Kubernetes controller extending Kubernetes' Service-based service discovery concept to work over all network interfaces of a Pod

With this toolset DANM is able to provide multiple separated network interfaces, the possibility to use different networking back ends and advanced IPAM features for the pods.

Flannel

[Flannel](#) is a very simple overlay network that satisfies the Kubernetes requirements. Many people have reported success with Flannel and Kubernetes.

Google Compute Engine (GCE)

For the Google Compute Engine cluster configuration scripts, [advanced routing](#) is used to assign each VM a subnet (default is /24 - 254 IPs). Any traffic bound for that subnet will be routed directly to the VM by the GCE network fabric. This is in addition to the "main" IP address assigned to the VM, which is NAT'ed for outbound internet access. A linux bridge (called `cbr 0`) is configured to exist on that subnet, and is passed to docker's `--bridge` flag.

Docker is started with:

```
DOCKER_OPTS="--bridge=cbr0 --iptables=false --ip-masq=false"
```

This bridge is created by Kubelet (controlled by the `--network-plugin=kubenet` flag) according to the Node's `.spec.podCIDR`.

Docker will now allocate IPs from the `cbr-cidr` block. Containers can reach each other and Nodes over the `cbr0` bridge. Those IPs are all routable within the GCE project network.

GCE itself does not know anything about these IPs, though, so it will not NAT them for outbound internet traffic. To achieve that an iptables rule is used to masquerade (aka SNAT - to make it seem as if packets came from

the Node itself) traffic that is bound for IPs outside the GCE project network (10.0.0.0/8).

```
iptables -t nat -A POSTROUTING ! -d 10.0.0.0/8 -o eth0 -j  
MASQUERADE
```

Lastly IP forwarding is enabled in the kernel (so the kernel will process packets for bridged containers):

```
sysctl net.ipv4.ip_forward=1
```

The result of all this is that all Pods can reach each other and can egress traffic to the internet.

Jaguar

[Jaguar](#) is an open source solution for Kubernetes's network based on OpenDaylight. Jaguar provides overlay network using vxlan and Jaguar CNIPlugin provides one IP address per pod.

k-vswitch

[k-vswitch](#) is a simple Kubernetes networking plugin based on [Open vSwitch](#). It leverages existing functionality in Open vSwitch to provide a robust networking plugin that is easy-to-operate, performant and secure.

Knitter

[Knitter](#) is a network solution which supports multiple networking in Kubernetes. It provides the ability of tenant management and network management. Knitter includes a set of end-to-end NFV container networking solutions besides multiple network planes, such as keeping IP address for applications, IP address migration, etc.

Kube-OVN

[Kube-OVN](#) is an OVN-based kubernetes network fabric for enterprises. With the help of OVN/OVS, it provides some advanced overlay network features like subnet, QoS, static IP allocation, traffic mirroring, gateway, openflow-based network policy and service proxy.

Kube-router

[Kube-router](#) is a purpose-built networking solution for Kubernetes that aims to provide high performance and operational simplicity. Kube-router provides a Linux [LVS/IPVS](#)-based service proxy, a Linux kernel forwarding-based pod-to-pod networking solution with no overlays, and iptables/ipset-based network policy enforcer.

L2 networks and linux bridging

If you have a "dumb" L2 network, such as a simple switch in a "bare-metal" environment, you should be able to do something similar to the above GCE setup. Note that these instructions have only been tried very casually - it seems to work, but has not been thoroughly tested. If you use this technique and perfect the process, please let us know.

Follow the "With Linux Bridge devices" section of [this very nice tutorial](#) from Lars Kellogg-Stedman.

Multus (a Multi Network plugin)

[Multus](#) is a Multi CNI plugin to support the Multi Networking feature in Kubernetes using CRD based network objects in Kubernetes.

Multus supports all [reference plugins](#) (eg. [Flannel](#), [DHCP](#), [Macvlan](#)) that implement the CNI specification and 3rd party plugins (eg. [Calico](#), [Weave](#), [Cilium](#), [Contiv](#)). In addition to it, Multus supports [SRIOV](#), [DPDK](#), [OVS-DPDK](#) & [VPP](#) workloads in Kubernetes with both cloud native and NFV based applications in Kubernetes.

OVN4NFV-K8s-Plugin (OVN based CNI controller & plugin)

[OVN4NFV-K8S-Plugin](#) is OVN based CNI controller plugin to provide cloud native based Service function chaining(SFC), Multiple OVN overlay networking, dynamic subnet creation, dynamic creation of virtual networks, VLAN Provider network, Direct provider network and pluggable with other Multi-network plugins, ideal for edge based cloud native workloads in Multi-cluster networking

NSX-T

[VMware NSX-T](#) is a network virtualization and security platform. NSX-T can provide network virtualization for a multi-cloud and multi-hypervisor environment and is focused on emerging application frameworks and architectures that have heterogeneous endpoints and technology stacks. In addition to vSphere hypervisors, these environments include other hypervisors such as KVM, containers, and bare metal.

[NSX-T Container Plug-in \(NCP\)](#) provides integration between NSX-T and container orchestrators such as Kubernetes, as well as integration between NSX-T and container-based CaaS/PaaS platforms such as Pivotal Container Service (PKS) and OpenShift.

Nuage Networks VCS (Virtualized Cloud Services)

[Nuage](#) provides a highly scalable policy-based Software-Defined Networking (SDN) platform. Nuage uses the open source Open vSwitch for the data plane along with a feature rich SDN Controller built on open standards.

The Nuage platform uses overlays to provide seamless policy-based networking between Kubernetes Pods and non-Kubernetes environments (VMs and bare metal servers). Nuage's policy abstraction model is designed with applications in mind and makes it easy to declare fine-grained policies for applications. The platform's real-time analytics engine enables visibility and security monitoring for Kubernetes applications.

OpenVSwitch

[OpenVSwitch](#) is a somewhat more mature but also complicated way to build an overlay network. This is endorsed by several of the "Big Shops" for networking.

OVN (Open Virtual Networking)

OVN is an opensource network virtualization solution developed by the Open vSwitch community. It lets one create logical switches, logical routers, stateful ACLs, load-balancers etc to build different virtual networking topologies. The project has a specific Kubernetes plugin and documentation at [ovn-kubernetes](#).

Romana

[Romana](#) is an open source network and security automation solution that lets you deploy Kubernetes without an overlay network. Romana supports Kubernetes [Network Policy](#) to provide isolation across network namespaces.

Weave Net from Weaveworks

[Weave Net](#) is a resilient and simple to use network for Kubernetes and its hosted applications. Weave Net runs as a [CNI plug-in](#) or stand-alone. In either version, it doesn't require any configuration or extra code to run, and in both cases, the network provides one IP address per pod - as is standard for Kubernetes.

What's next

The early design of the networking model and its rationale, and some future plans are described in more detail in the [networking design document](#).

Feedback

Was this page helpful?

Yes No

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

Last modified December 16, 2020 at 2:52 AM PST: [Space screwed up formatting \(c5006a6f1\)](#)

[Edit this page](#) [Create child page](#) [Create an issue](#)

- [The Kubernetes network model](#)
- [How to implement the Kubernetes networking model](#)
 - [ACI](#)
 - [Antrea](#)
 - [AOS from Apstra](#)
 - [AWS VPC CNI for Kubernetes](#)
 - [Azure CNI for Kubernetes](#)
 - [Big Cloud Fabric from Big Switch Networks](#)
 - [Calico](#)
 - [Cilium](#)
 - [CNI-Genie from Huawei](#)
 - [cni-ipvlan-vpc-k8s](#)
 - [Coil](#)
 - [Contiv](#)
 - [Contrail / Tungsten Fabric](#)
 - [DANM](#)
 - [Flannel](#)
 - [Google Compute Engine \(GCE\)](#)
 - [Jaguar](#)
 - [k-vswitch](#)
 - [Knitter](#)
 - [Kube-OVN](#)
 - [Kube-router](#)
 - [L2 networks and linux bridging](#)
 - [Multus \(a Multi Network plugin\)](#)
 - [OVN4NFV-K8s-Plugin \(OVN based CNI controller & plugin\)](#)
 - [NSX-T](#)
 - [Nuage Networks VCS \(Virtualized Cloud Services\)](#)
 - [OpenVSwitch](#)
 - [OVN \(Open Virtual Networking\)](#)
 - [Romana](#)
 - [Weave Net from Weaveworks](#)
- [What's next](#)

Logging Architecture

Application logs can help you understand what is happening inside your application. The logs are particularly useful for debugging problems and monitoring cluster activity. Most modern applications have some kind of logging mechanism; as such, most container engines are likewise designed to support some kind of logging. The easiest and most embraced logging method for containerized applications is to write to the standard output and standard error streams.

However, the native functionality provided by a container engine or runtime is usually not enough for a complete logging solution. For example, if a container crashes, a pod is evicted, or a node dies, you'll usually still want to

access your application's logs. As such, logs should have a separate storage and lifecycle independent of nodes, pods, or containers. This concept is called *cluster-level-logging*. Cluster-level logging requires a separate backend to store, analyze, and query logs. Kubernetes provides no native storage solution for log data, but you can integrate many existing logging solutions into your Kubernetes cluster.

Cluster-level logging architectures are described in assumption that a logging backend is present inside or outside of your cluster. If you're not interested in having cluster-level logging, you might still find the description of how logs are stored and handled on the node to be useful.

Basic logging in Kubernetes

In this section, you can see an example of basic logging in Kubernetes that outputs data to the standard output stream. This demonstration uses a pod specification with a container that writes some text to standard output once per second.

[debug/counter-pod.yaml](#)



```
apiVersion: v1
kind: Pod
metadata:
  name: counter
spec:
  containers:
    - name: count
      image: busybox
      args: ["/bin/sh", "-c",
              'i=0; while true; do echo "$i: $(date)"; i=$((i+1));
sleep 1; done']
```

To run this pod, use the following command:

```
kubectl apply -f https://k8s.io/examples/debug/counter-pod.yaml
```

The output is:

```
pod/counter created
```

To fetch the logs, use the `kubectl logs` command, as follows:

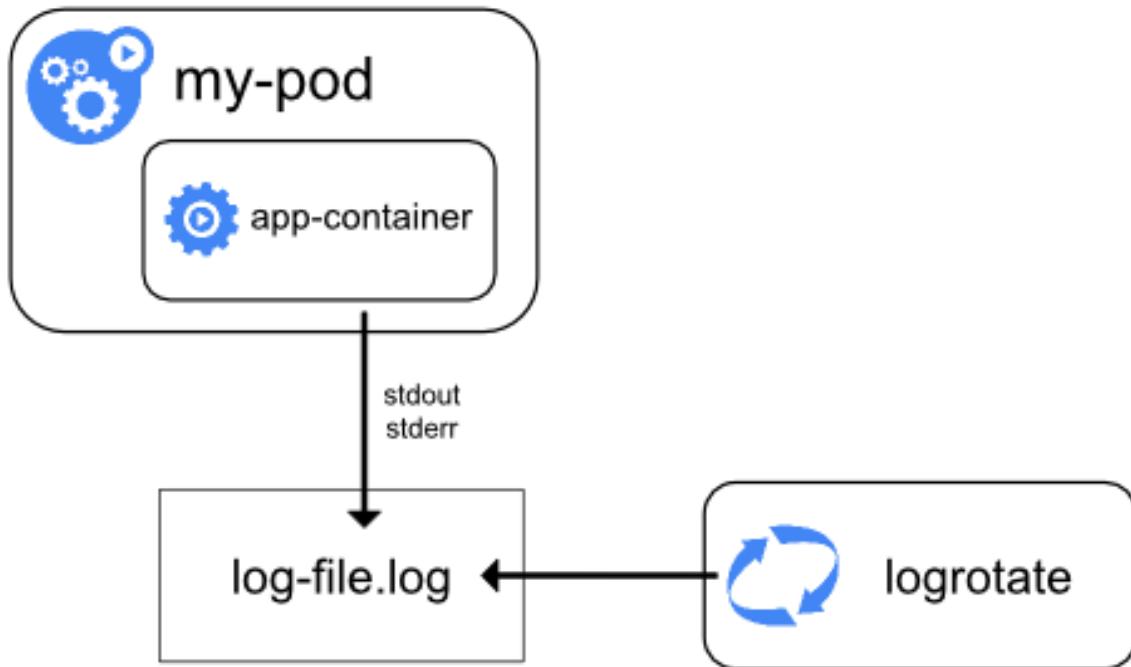
```
kubectl logs counter
```

The output is:

```
0: Mon Jan  1 00:00:00 UTC 2001
1: Mon Jan  1 00:00:01 UTC 2001
2: Mon Jan  1 00:00:02 UTC 2001
...
```

You can use `kubectl logs` to retrieve logs from a previous instantiation of a container with `--previous` flag, in case the container has crashed. If your pod has multiple containers, you should specify which container's logs you want to access by appending a container name to the command. See the [kubectl logs documentation](#) for more details.

Logging at the node level



Everything a containerized application writes to `stdout` and `stderr` is handled and redirected somewhere by a container engine. For example, the Docker container engine redirects those two streams to [a logging driver](#), which is configured in Kubernetes to write to a file in json format.

Note: The Docker json logging driver treats each line as a separate message. When using the Docker logging driver, there is no direct support for multi-line messages. You need to handle multi-line messages at the logging agent level or higher.

By default, if a container restarts, the `kubelet` keeps one terminated container with its logs. If a pod is evicted from the node, all corresponding containers are also evicted, along with their logs.

An important consideration in node-level logging is implementing log rotation, so that logs don't consume all available storage on the node. Kubernetes currently is not responsible for rotating logs, but rather a deployment tool should set up a solution to address that. For example, in Kubernetes clusters, deployed by the `kube-up.sh` script, there is a [logrotate](#) tool configured to run each hour. You can also set up a container runtime to rotate application's logs automatically, for example by using Docker's `log-opt`. In the `kube-up.sh` script, the latter approach is used for COS image

on GCP, and the former approach is used in any other environment. In both cases, by default rotation is configured to take place when log file exceeds 10MB.

As an example, you can find detailed information about how `kube-up.sh` sets up logging for COS image on GCP in the corresponding [script](#).

When you run `kubectl logs` as in the basic logging example, the kubelet on the node handles the request and reads directly from the log file, returning the contents in the response.

Note: Currently, if some external system has performed the rotation, only the contents of the latest log file will be available through `kubectl logs`. E.g. if there's a 10MB file, `logrotate` performs the rotation and there are two files, one 10MB in size and one empty, `kubectl logs` will return an empty response.

System component logs

There are two types of system components: those that run in a container and those that do not run in a container. For example:

- The Kubernetes scheduler and `kube-proxy` run in a container.
- The `kubelet` and container runtime, for example Docker, do not run in containers.

On machines with `systemd`, the `kubelet` and container runtime write to `journald`. If `systemd` is not present, they write to `.log` files in the `/var/log` directory. System components inside containers always write to the `/var/log` directory, bypassing the default logging mechanism. They use the [`klog`](#) logging library. You can find the conventions for logging severity for those components in the [development docs on logging](#).

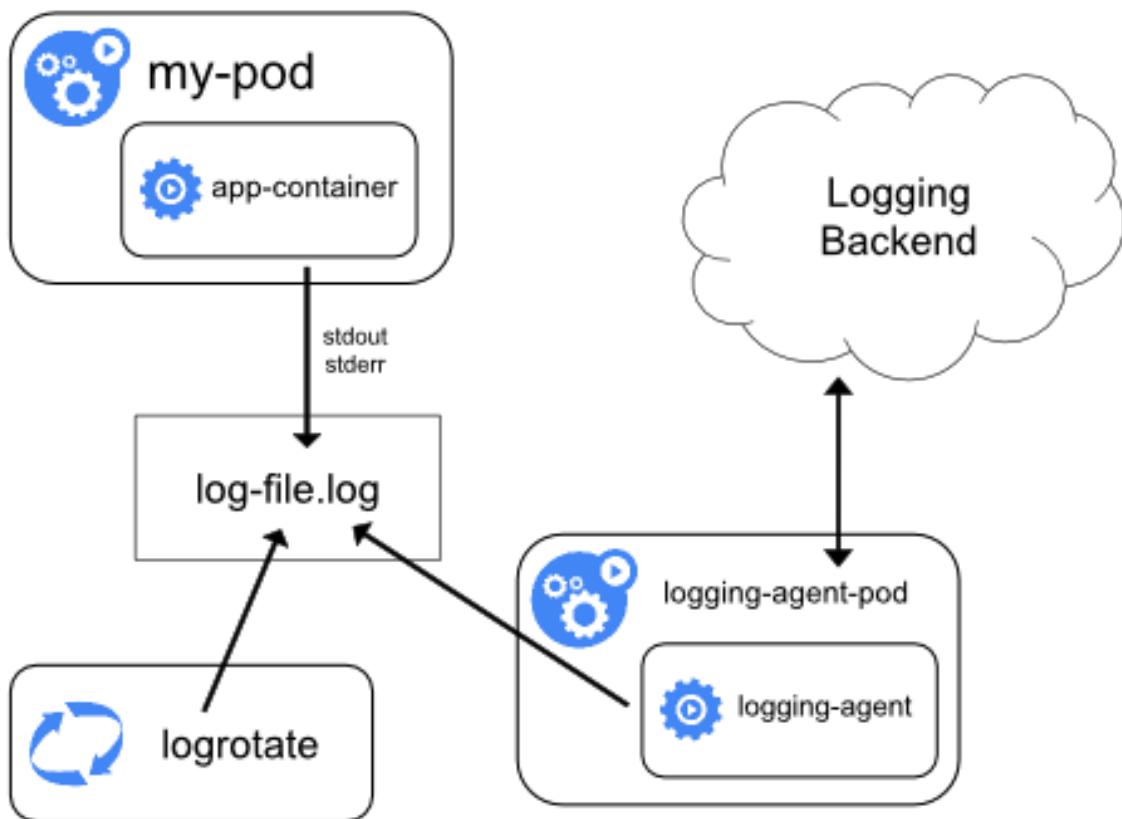
Similarly to the container logs, system component logs in the `/var/log` directory should be rotated. In Kubernetes clusters brought up by the `kube-up.sh` script, those logs are configured to be rotated by the `logrotate` tool daily or once the size exceeds 100MB.

Cluster-level logging architectures

While Kubernetes does not provide a native solution for cluster-level logging, there are several common approaches you can consider. Here are some options:

- Use a node-level logging agent that runs on every node.
- Include a dedicated sidecar container for logging in an application pod.
- Push logs directly to a backend from within an application.

Using a node logging agent



You can implement cluster-level logging by including a node-level logging agent on each node. The logging agent is a dedicated tool that exposes logs or pushes logs to a backend. Commonly, the logging agent is a container that has access to a directory with log files from all of the application containers on that node.

Because the logging agent must run on every node, it's common to implement it as either a DaemonSet replica, a manifest pod, or a dedicated native process on the node. However the latter two approaches are deprecated and highly discouraged.

Using a node-level logging agent is the most common and encouraged approach for a Kubernetes cluster, because it creates only one agent per node, and it doesn't require any changes to the applications running on the node. However, node-level logging only works for applications' standard output and standard error.

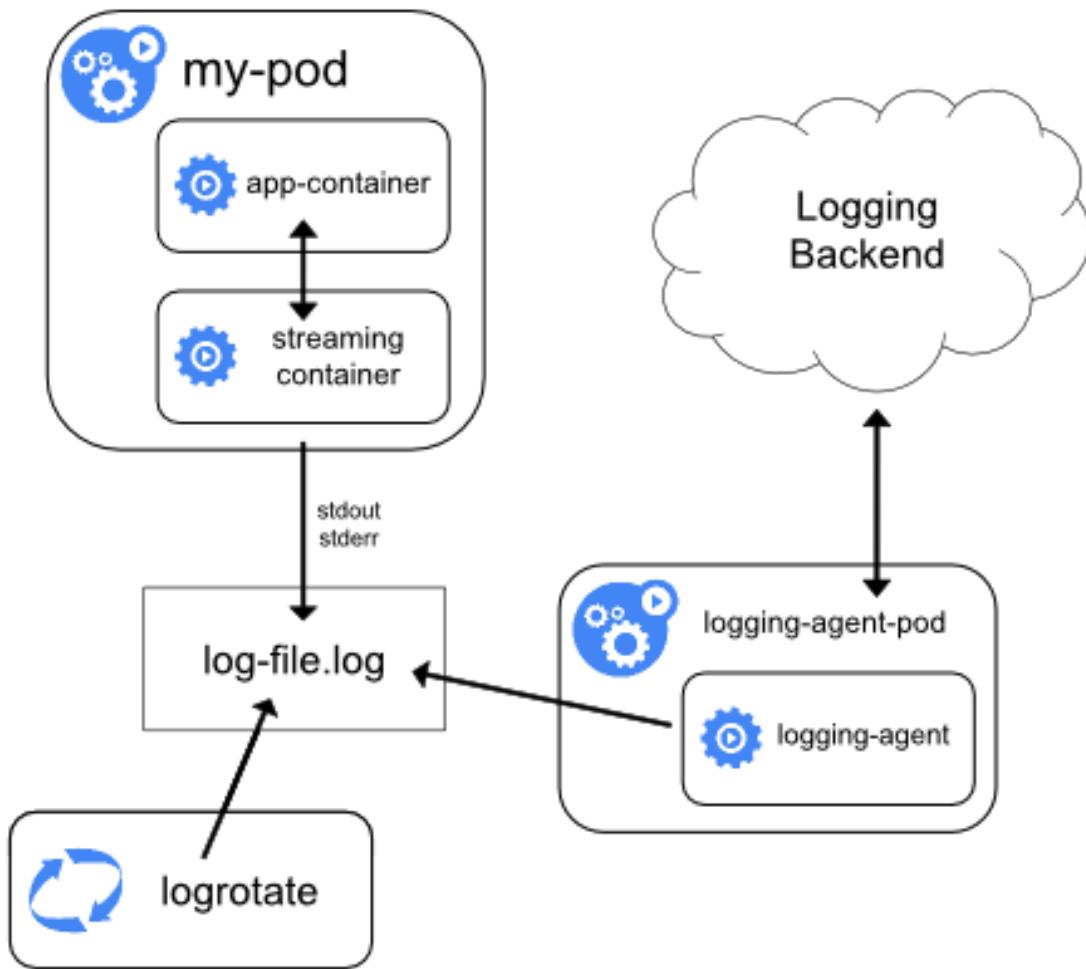
Kubernetes doesn't specify a logging agent, but two optional logging agents are packaged with the Kubernetes release: [Stackdriver Logging](#) for use with Google Cloud Platform, and [Elasticsearch](#). You can find more information and instructions in the dedicated documents. Both use [fluentd](#) with custom configuration as an agent on the node.

Using a sidecar container with the logging agent

You can use a sidecar container in one of the following ways:

- The sidecar container streams application logs to its own `stdout`.
- The sidecar container runs a logging agent, which is configured to pick up logs from an application container.

Streaming sidecar container



By having your sidecar containers stream to their own `stdout` and `stderr` streams, you can take advantage of the kubelet and the logging agent that already run on each node. The sidecar containers read logs from a file, a socket, or the `journald`. Each individual sidecar container prints log to its own `stdout` or `stderr` stream.

This approach allows you to separate several log streams from different parts of your application, some of which can lack support for writing to `stdout` or `stderr`. The logic behind redirecting logs is minimal, so it's hardly a significant overhead. Additionally, because `stdout` and `stderr` are handled by the kubelet, you can use built-in tools like `kubectl logs`.

Consider the following example. A pod runs a single container, and the container writes to two different log files, using two different formats. Here's a configuration file for the Pod:

[admin/logging/two-files-counter-pod.yaml](#)
[View]

```
apiVersion: v1
kind: Pod
metadata:
  name: counter
spec:
  containers:
    - name: count
      image: busybox
      args:
        - /bin/sh
        - -c
        - >
          i=0;
          while true;
          do
            echo "$i: $(date)" >> /var/log/1.log;
            echo "$(date) INFO $i" >> /var/log/2.log;
            i=$((i+1));
            sleep 1;
          done
  volumeMounts:
    - name: varlog
      mountPath: /var/log
  volumes:
    - name: varlog
      emptyDir: {}
```

It would be a mess to have log entries of different formats in the same log stream, even if you managed to redirect both components to the `stdout` stream of the container. Instead, you could introduce two sidecar containers. Each sidecar container could tail a particular log file from a shared volume and then redirect the logs to its own `stdout` stream.

Here's a configuration file for a pod that has two sidecar containers:

[admin/logging/two-files-counter-pod-streaming-sidecar.yaml](#)
[View]

```
apiVersion: v1
kind: Pod
metadata:
  name: counter
spec:
  containers:
    - name: count
```

```



```

Now when you run this pod, you can access each log stream separately by running the following commands:

```
kubectl logs counter count-log-1
```

```
0: Mon Jan  1 00:00:00 UTC 2001
1: Mon Jan  1 00:00:01 UTC 2001
2: Mon Jan  1 00:00:02 UTC 2001
...
```

```
kubectl logs counter count-log-2
```

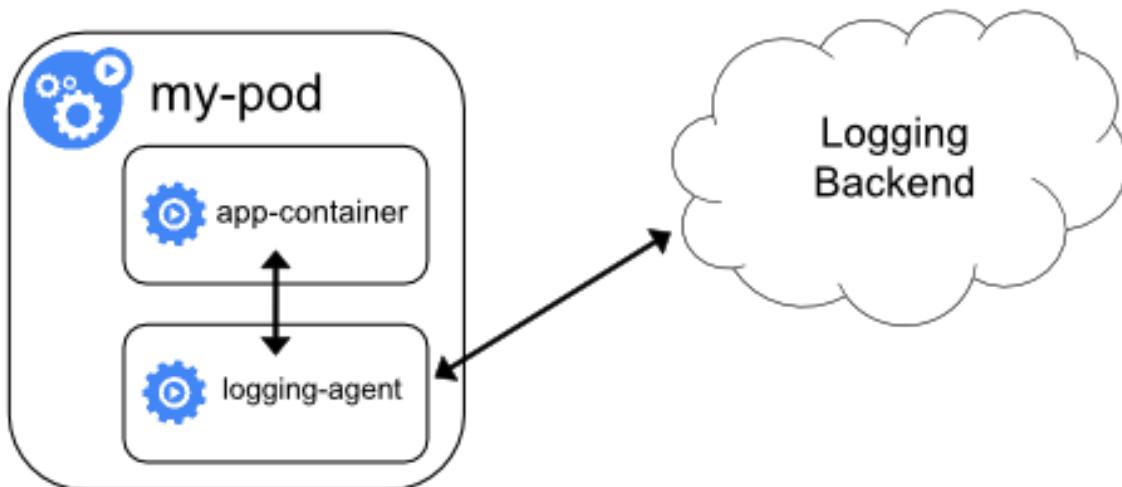
```
Mon Jan  1 00:00:00 UTC 2001 INFO 0
Mon Jan  1 00:00:01 UTC 2001 INFO 1
Mon Jan  1 00:00:02 UTC 2001 INFO 2
...
```

The node-level agent installed in your cluster picks up those log streams automatically without any further configuration. If you like, you can configure the agent to parse log lines depending on the source container.

Note, that despite low CPU and memory usage (order of couple of millicores for cpu and order of several megabytes for memory), writing logs to a file and then streaming them to `stdout` can double disk usage. If you have an application that writes to a single file, it's generally better to set `/dev/stdout` as destination rather than implementing the streaming sidecar container approach.

Sidecar containers can also be used to rotate log files that cannot be rotated by the application itself. An example of this approach is a small container running logrotate periodically. However, it's recommended to use `stdout` and `stderr` directly and leave rotation and retention policies to the kubelet.

Sidecar container with a logging agent



If the node-level logging agent is not flexible enough for your situation, you can create a sidecar container with a separate logging agent that you have configured specifically to run with your application.

Note: Using a logging agent in a sidecar container can lead to significant resource consumption. Moreover, you won't be able to access those logs using `kubectl logs` command, because they are not controlled by the kubelet.

As an example, you could use [Stackdriver](#), which uses fluentd as a logging agent. Here are two configuration files that you can use to implement this approach. The first file contains a [ConfigMap](#) to configure fluentd.

[admin/logging/fluentd-sidecar-config.yaml](#)
□

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: fluentd-config
```

```

data:
  fluentd.conf: |
    <source>
      type tail
      format none
      path /var/log/1.log
      pos_file /var/log/1.log.pos
      tag count.format1
    </source>

    <source>
      type tail
      format none
      path /var/log/2.log
      pos_file /var/log/2.log.pos
      tag count.format2
    </source>

    <match **>
      type google_cloud
    </match>

```

Note: The configuration of fluentd is beyond the scope of this article. For information about configuring fluentd, see the [official fluentd documentation](#).

The second file describes a pod that has a sidecar container running fluentd. The pod mounts a volume where fluentd can pick up its configuration data.

[admin/logging/two-files-counter-pod-agent-sidecar.yaml](#)



```

apiVersion: v1
kind: Pod
metadata:
  name: counter
spec:
  containers:
  - name: count
    image: busybox
    args:
    - /bin/sh
    - -C
    - >
      i=0;
      while true;
      do
        echo "$i: $(date)" >> /var/log/1.log;
        echo "$(date) INFO $i" >> /var/log/2.log;
        i=$((i+1));
        sleep 1;
      done

```

```

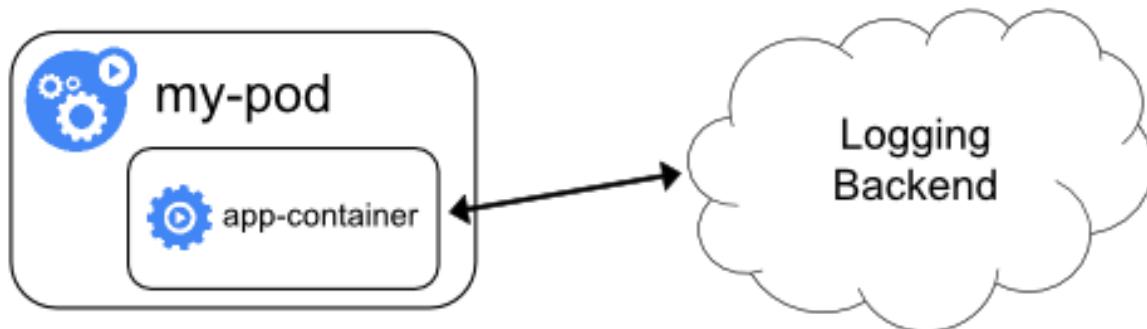
volumeMounts:
- name: varlog
  mountPath: /var/log
- name: count-agent
  image: k8s.gcr.io/fluentd-gcp:1.30
env:
- name: FLUENTD_ARGS
  value: -c /etc/fluentd-config/fluentd.conf
volumeMounts:
- name: varlog
  mountPath: /var/log
- name: config-volume
  mountPath: /etc/fluentd-config
volumes:
- name: varlog
  emptyDir: {}
- name: config-volume
  configMap:
    name: fluentd-config

```

After some time you can find log messages in the Stackdriver interface.

Remember, that this is just an example and you can actually replace fluentd with any logging agent, reading from any source inside an application container.

Exposing logs directly from the application



You can implement cluster-level logging by exposing or pushing logs directly from every application; however, the implementation for such a logging mechanism is outside the scope of Kubernetes.

Feedback

Was this page helpful?

Yes No

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

Last modified December 07, 2020 at 11:24 AM PST: [Add a missing period in logging.md \(b1f91de70\)](#)

[Edit this page](#) [Create child page](#) [Create an issue](#)

- [Basic logging in Kubernetes](#)
- [Logging at the node level](#)
 - [System component logs](#)
- [Cluster-level logging architectures](#)
 - [Using a node logging agent](#)
 - [Using a sidecar container with the logging agent](#)
 - [Exposing logs directly from the application](#)

Metrics For Kubernetes System Components

System component metrics can give a better look into what is happening inside them. Metrics are particularly useful for building dashboards and alerts.

Kubernetes components emit metrics in [Prometheus format](#). This format is structured plain text, designed so that people and machines can both read it.

Metrics in Kubernetes

In most cases metrics are available on `/metrics` endpoint of the HTTP server. For components that doesn't expose endpoint by default it can be enabled using `--bind-address` flag.

Examples of those components:

- [kube-controller-manager](#)
- [kube-proxy](#)
- [kube-apiserver](#)
- [kube-scheduler](#)
- [kublet](#)

In a production environment you may want to configure [Prometheus Server](#) or some other metrics scraper to periodically gather these metrics and make them available in some kind of time series database.

Note that [kublet](#) also exposes metrics in `/metrics/cadvisor`, `/metrics/resource` and `/metrics/probes` endpoints. Those metrics do not have same lifecycle.

If your cluster uses [RBAC](#), reading metrics requires authorization via a user, group or ServiceAccount with a ClusterRole that allows accessing `/metrics`. For example:

```
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRole
metadata:
  name: prometheus
rules:
  - nonResourceURLs:
    - "/metrics"
  verbs:
    - get
```

Metric lifecycle

Alpha metric | *Stable metric* | *Deprecated metric* | *Hidden metric* | *Deleted metric*

Alpha metrics have no stability guarantees. These metrics can be modified or deleted at any time.

Stable metrics are guaranteed to not change. This means:

- A stable metric without a deprecated signature will not be deleted or renamed
- A stable metric's type will not be modified

Deprecated metrics are slated for deletion, but are still available for use. These metrics include an annotation about the version in which they became deprecated.

For example:

- Before deprecation

```
# HELP some_counter this counts things
# TYPE some_counter counter
some_counter 0
```

- After deprecation

```
# HELP some_counter (Deprecated since 1.15.0) this counts
things
# TYPE some_counter counter
some_counter 0
```

Hidden metrics are no longer published for scraping, but are still available for use. To use a hidden metric, please refer to the [Show hidden metrics](#) section.

Deleted metrics are no longer published and cannot be used.

Show hidden metrics

As described above, admins can enable hidden metrics through a command-line flag on a specific binary. This intends to be used as an escape hatch for admins if they missed the migration of the metrics deprecated in the last release.

The flag `show-hidden-metrics-for-version` takes a version for which you want to show metrics deprecated in that release. The version is expressed as `x.y`, where `x` is the major version, `y` is the minor version. The patch version is not needed even though a metrics can be deprecated in a patch release, the reason for that is the metrics deprecation policy runs against the minor release.

The flag can only take the previous minor version as its value. All metrics hidden in previous will be emitted if admins set the previous version to `show-hidden-metrics-for-version`. The too old version is not allowed because this violates the metrics deprecated policy.

Take metric `A` as an example, here assumed that `A` is deprecated in `1.n`. According to metrics deprecated policy, we can reach the following conclusion:

- In release `1.n`, the metric is deprecated, and it can be emitted by default.
- In release `1.n+1`, the metric is hidden by default and it can be emitted by command line `show-hidden-metrics-for-version=1.n`.
- In release `1.n+2`, the metric should be removed from the codebase. No escape hatch anymore.

If you're upgrading from release `1.12` to `1.13`, but still depend on a metric `A` deprecated in `1.12`, you should set hidden metrics via command line: `--show-hidden-metrics=1.12` and remember to remove this metric dependency before upgrading to `1.14`

Disable accelerator metrics

The kubelet collects accelerator metrics through cAdvisor. To collect these metrics, for accelerators like NVIDIA GPUs, kubelet held an open handle on the driver. This meant that in order to perform infrastructure changes (for example, updating the driver), a cluster administrator needed to stop the kubelet agent.

The responsibility for collecting accelerator metrics now belongs to the vendor rather than the kubelet. Vendors must provide a container that collects metrics and exposes them to the metrics service (for example, Prometheus).

The [`DisableAcceleratorUsageMetrics` feature gate](#) disables metrics collected by the kubelet, with a [`timeline for enabling this feature by default`](#).

Component metrics

kube-controller-manager metrics

Controller manager metrics provide important insight into the performance and health of the controller manager. These metrics include common Go language runtime metrics such as go_routine count and controller specific metrics such as etcd request latencies or Cloudprovider (AWS, GCE, OpenStack) API latencies that can be used to gauge the health of a cluster.

Starting from Kubernetes 1.7, detailed Cloudprovider metrics are available for storage operations for GCE, AWS, Vsphere and OpenStack. These metrics can be used to monitor health of persistent volume operations.

For example, for GCE these metrics are called:

```
cloudprovider_gce_api_request_duration_seconds { request = "instance_list"}  
cloudprovider_gce_api_request_duration_seconds { request = "disk_insert"}  
cloudprovider_gce_api_request_duration_seconds { request = "disk_delete"}  
cloudprovider_gce_api_request_duration_seconds { request = "attach_disk"}  
cloudprovider_gce_api_request_duration_seconds { request = "detach_disk"}  
cloudprovider_gce_api_request_duration_seconds { request = "list_disk"}
```

kube-scheduler metrics

FEATURE STATE: Kubernetes v1.20 [alpha]

The scheduler exposes optional metrics that reports the requested resources and the desired limits of all running pods. These metrics can be used to build capacity planning dashboards, assess current or historical scheduling limits, quickly identify workloads that cannot schedule due to lack of resources, and compare actual usage to the pod's request.

The kube-scheduler identifies the resource [requests and limits](#) configured for each Pod; when either a request or limit is non-zero, the kube-scheduler reports a metrics timeseries. The time series is labelled by:

- namespace
- pod name
- the node where the pod is scheduled or an empty string if not yet scheduled
- priority
- the assigned scheduler for that pod
- the name of the resource (for example, cpu)
- the unit of the resource if known (for example, cores)

Once a pod reaches completion (has a `restartPolicy` of `Never` or `OnFailure` and is in the `Succeeded` or `Failed` pod phase, or has been deleted and all containers have a terminated state) the series is no longer reported since the scheduler is now free to schedule other pods to run. The two metrics are called `kube_pod_resource_request` and `kube_pod_resource_limit`.

The metrics are exposed at the HTTP endpoint `/metrics/resources` and require the same authorization as the `/metrics` endpoint on the scheduler. You must use the `--show-hidden-metrics-for-version=1.20` flag to expose these alpha stability metrics.

What's next

- Read about the [Prometheus text format](#) for metrics
- See the list of [stable Kubernetes metrics](#)
- Read about the [Kubernetes deprecation policy](#)

Feedback

Was this page helpful?

Yes No

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

Last modified November 28, 2020 at 11:07 AM PST: [Updating docs/concepts/cluster-administration/system-metrics \(e38868f47\)](#)
[Edit this page](#) [Create child page](#) [Create an issue](#)

- [Metrics in Kubernetes](#)
- [Metric lifecycle](#)
- [Show hidden metrics](#)
- [Disable accelerator metrics](#)
- [Component metrics](#)
 - [kube-controller-manager metrics](#)
 - [kube-scheduler metrics](#)
- [What's next](#)

System Logs

System component logs record events happening in cluster, which can be very useful for debugging. You can configure log verbosity to see more or less detail. Logs can be as coarse-grained as showing errors within a component, or as fine-grained as showing step-by-step traces of events (like HTTP access logs, pod state changes, controller actions, or scheduler decisions).

Klog

klog is the Kubernetes logging library. [klog](#) generates log messages for the Kubernetes system components.

For more information about klog configuration, see the [Command line tool reference](#).

An example of the klog native format:

```
I1025 00:15:15.525108      1 httplog.go:79] GET /api/v1/namespaces/kube-system/pods/metrics-server-v0.3.1-57c75779f-9p8wg: (1.512ms) 200 [pod_nanny/v0.0.0 (linux/amd64) kubernetes/$Format 10.56.1.19:51756]
```

Structured Logging

FEATURE STATE: Kubernetes v1.19 [alpha]

Warning:

Migration to structured log messages is an ongoing process. Not all log messages are structured in this version. When parsing log files, you must also handle unstructured log messages.

Log formatting and value serialization are subject to change.

Structured logging is a effort to introduce a uniform structure in log messages allowing for easy extraction of information, making logs easier and cheaper to store and process. New message format is backward compatible and enabled by default.

Format of structured logs:

```
<klog header> "<message>" <key1>=<value1> <key2>=<value2> ...
```

Example:

```
I1025 00:15:15.525108      1 controller_utils.go:116] "Pod status updated" pod="kube-system/kubedns" status="ready"
```

JSON log format

FEATURE STATE: Kubernetes v1.19 [alpha]

Warning:

JSON output does not support many standard klog flags. For list of unsupported klog flags, see the [Command line tool reference](#).

Not all logs are guaranteed to be written in JSON format (for example, during process start). If you intend to parse logs, make sure you can handle log lines that are not JSON as well.

Field names and JSON serialization are subject to change.

The --logging-format=json flag changes the format of logs from klog native format to JSON format. Example of JSON log format (pretty printed):

```
{  
  "ts": 1580306777.04728,  
  "v": 4,  
  "msg": "Pod status updated",  
  "pod": {  
    "name": "nginx-1",  
    "namespace": "default"  
  },  
  "status": "ready"  
}
```

Keys with special meaning:

- *ts* - timestamp as Unix time (required, float)
- *v* - verbosity (required, int, default 0)
- *err* - error string (optional, string)
- *msg* - message (required, string)

List of components currently supporting JSON format:

- [kube-controller-manager](#)
- [kube-apiserver](#)
- [kube-scheduler](#)
- [kubelet](#)

Log sanitization

FEATURE STATE: Kubernetes v1.20 [alpha]

Warning: Log sanitization might incur significant computation overhead and therefore should not be enabled in production.

The --experimental-logging-sanitization flag enables the klog sanitization filter. If enabled all log arguments are inspected for fields tagged as sensitive data (e.g. passwords, keys, tokens) and logging of these fields will be prevented.

List of components currently supporting log sanitization:

- *kube-controller-manager*
- *kube-apiserver*
- *kube-scheduler*

- *kubelet*

Note: The Log sanitization filter does not prevent user workload logs from leaking sensitive data.

Log verbosity level

The `-v` flag controls log verbosity. Increasing the value increases the number of logged events. Decreasing the value decreases the number of logged events. Increasing verbosity settings logs increasingly less severe events. A verbosity setting of 0 logs only critical events.

Log location

There are two types of system components: those that run in a container and those that do not run in a container. For example:

- The Kubernetes scheduler and kube-proxy run in a container.
- The kubelet and container runtime, for example Docker, do not run in containers.

On machines with `systemd`, the kubelet and container runtime write to `journald`. Otherwise, they write to `.log` files in the `/var/log` directory. System components inside containers always write to `.log` files in the `/var/log` directory, bypassing the default logging mechanism. Similar to the container logs, you should rotate system component logs in the `/var/log` directory. In Kubernetes clusters created by the `kube-up.sh` script, log rotation is configured by the `logrotate` tool. The `logrotate` tool rotates logs daily, or once the log size is greater than 100MB.

What's next

- Read about the [Kubernetes Logging Architecture](#)
- Read about [Structured Logging](#)
- Read about the [Conventions for logging severity](#)

Feedback

Was this page helpful?

Yes No

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

Last modified November 24, 2020 at 9:10 AM PST: [Small wording update on Log sanitization docs \(1de3ac5a2\)](#)

- [Klog](#)
 - [Structured Logging](#)
 - [JSON log format](#)
 - [Log sanitization](#)
 - [Log verbosity level](#)
 - [Log location](#)
- [What's next](#)

Garbage collection for container images

Garbage collection is a helpful function of kubelet that will clean up unused [images](#) and unused [containers](#). Kubelet will perform garbage collection for containers every minute and garbage collection for images every five minutes.

External garbage collection tools are not recommended as these tools can potentially break the behavior of kubelet by removing containers expected to exist.

Image Collection

Kubernetes manages lifecycle of all images through `imageManager`, with the cooperation of `cadvisor`.

The policy for garbage collecting images takes two factors into consideration: `HighThresholdPercent` and `LowThresholdPercent`. Disk usage above the high threshold will trigger garbage collection. The garbage collection will delete least recently used images until the low threshold has been met.

Container Collection

The policy for garbage collecting containers considers three user-defined variables. `MinAge` is the minimum age at which a container can be garbage collected. `MaxPerPodContainer` is the maximum number of dead containers every single pod (UID, container name) pair is allowed to have. `MaxContainers` is the maximum number of total dead containers. These variables can be individually disabled by setting `MinAge` to zero and setting `MaxPerPodContainer` and `MaxContainers` respectively to less than zero.

Kubelet will act on containers that are unidentified, deleted, or outside of the boundaries set by the previously mentioned flags. The oldest containers will generally be removed first. `MaxPerPodContainer` and `MaxContainer` may potentially conflict with each other in situations where retaining the maximum number of containers per pod (`MaxPerPodContainer`) would go outside the allowable range of global dead containers (`MaxContainers`). Max

PerPodContainer would be adjusted in this situation: A worst case scenario would be to downgrade *MaxPerPodContainer* to 1 and evict the oldest containers. Additionally, containers owned by pods that have been deleted are removed once they are older than *MinAge*.

Containers that are not managed by kubelet are not subject to container garbage collection.

User Configuration

You can adjust the following thresholds to tune image garbage collection with the following kubelet flags :

1. *image-gc-high-threshold*, the percent of disk usage which triggers image garbage collection. Default is 85%.
2. *image-gc-low-threshold*, the percent of disk usage to which image garbage collection attempts to free. Default is 80%.

You can customize the garbage collection policy through the following kubelet flags:

1. *minimum-container-ttl-duration*, minimum age for a finished container before it is garbage collected. Default is 0 minute, which means every finished container will be garbage collected.
2. *maximum-dead-containers-per-container*, maximum number of old instances to be retained per container. Default is 1.
3. *maximum-dead-containers*, maximum number of old instances of containers to retain globally. Default is -1, which means there is no global limit.

Containers can potentially be garbage collected before their usefulness has expired. These containers can contain logs and other data that can be useful for troubleshooting. A sufficiently large value for *maximum-dead-containers-per-container* is highly recommended to allow at least 1 dead container to be retained per expected container. A larger value for *maximum-dead-containers* is also recommended for a similar reason. See [this issue](#) for more details.

Deprecation

Some kubelet Garbage Collection features in this doc will be replaced by kubelet eviction in the future.

Including:

Existing Flag	New Flag	Rationale
--image-gc-high-threshold	--eviction-hard or --eviction-soft	existing eviction signals can trigger image garbage collection
--image-gc-low-threshold	--eviction-minimum-reclaim	eviction reclaims achieve the same behavior

Existing Flag	New Flag	Rationale
--maximum-dead-containers		deprecated once old logs are stored outside of container's context
--maximum-dead-containers-per-container		deprecated once old logs are stored outside of container's context
--minimum-container-ttl-duration		deprecated once old logs are stored outside of container's context
--low-diskspace-threshold-mb	--eviction-hard or --eviction-soft	eviction generalizes disk thresholds to other resources
--outofdisk-transition-frequency	--eviction-pressure-transition-period	eviction generalizes disk pressure transition to other resources

What's next

See [Configuring Out Of Resource Handling](#) for more details.

Feedback

Was this page helpful?

Yes No

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

Last modified December 07, 2020 at 5:07 PM PST: [Retitle "Configuring kubelet Garbage Collection" \(7f45bcc3a\)](#)

[Edit this page](#) [Create child page](#) [Create an issue](#)

- [Image Collection](#)
- [Container Collection](#)
- [User Configuration](#)
- [Deprecation](#)
- [What's next](#)

Proxies in Kubernetes

This page explains proxies used with Kubernetes.

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, TCP and SCTP
- does not understand HTTP
- provides load balancing
- is just used to reach services

4. A Proxy/Load-balancer in front of apiserver(s):

- existence and implementation varies from cluster to cluster (e.g. nginx)
- sits between all clients and one or more apiservers
- acts as load balancer if there are several apiservers.

5. Cloud Load Balancers on external services:

- are provided by some cloud providers (e.g. AWS ELB, Google Cloud Load Balancer)
- are created automatically when the Kubernetes service has type LoadBalancer
- usually supports UDP/TCP only
- SCTP support is up to the load balancer implementation of the cloud provider
- implementation varies by cloud provider.

Kubernetes users will typically not need to worry about anything other than the first two types. The cluster admin will typically ensure that the latter types are setup correctly.

Requesting redirects

Proxies have replaced redirect capabilities. Redirects have been deprecated.

Feedback

Was this page helpful?

Yes No

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

*Last modified May 30, 2020 at 3:10 PM PST: [add en pages \(ecc27bbbe\)](#)
[Edit this page](#) [Create child page](#) [Create an issue](#)*

- [Proxies](#)
- [Requesting redirects](#)

API Priority and Fairness

FEATURE STATE: Kubernetes v1.20 [beta]

Controlling the behavior of the Kubernetes API server in an overload situation is a key task for cluster administrators. The [kube-apiserver](#) has some controls available (i.e. the `--max-requests-inflight` and `--max-mutating-requests-inflight` command-line flags) to limit the amount of outstanding work that will be accepted, preventing a flood of inbound requests from overloading and potentially crashing the API server, but these flags are not enough to ensure that the most important requests get through in a period of high traffic.

The API Priority and Fairness feature (APF) is an alternative that improves upon aforementioned max-inflight limitations. APF classifies and isolates requests in a more fine-grained way. It also introduces a limited amount of queuing, so that no requests are rejected in cases of very brief bursts. Requests are dispatched from queues using a fair queuing technique so that, for example, a poorly-behaved [controller](#) need not starve others (even at the same priority level).

Caution: Requests classified as "long-running" — primarily watches — are not subject to the API Priority and Fairness filter. This is also true for the `--max-requests-inflight` flag without the API Priority and Fairness feature enabled.

Enabling/Disabling API Priority and Fairness

The API Priority and Fairness feature is controlled by a feature gate and is enabled by default. See [Feature Gates](#) for a general explanation of feature gates and how to enable and disable them. The name of the feature gate for APF is "APIPriorityAndFairness". This feature also involves an [API Group](#) with: (a) a v1alpha1 version, disabled by default, and (b) a v1beta1 version, enabled by default. You can disable the feature gate and API group v1beta1 version by adding the following command-line flags to your kube-apiserver invocation:

```
kube-apiserver \
--feature-gates=APIPriorityAndFairness=false \
--runtime-config=flowcontrol.apiserver.k8s.io/v1beta1=false \
# and other flags as usual
```

Alternatively, you can enable the v1alpha1 version of the API group with --runtime-config=flowcontrol.apiserver.k8s.io/v1beta1=true.

The command-line flag --enable-priority-and-fairness=false will disable the API Priority and Fairness feature, even if other flags have enabled it.

Concepts

There are several distinct features involved in the API Priority and Fairness feature. Incoming requests are classified by attributes of the request using FlowSchemas, and assigned to priority levels. Priority levels add a degree of isolation by maintaining separate concurrency limits, so that requests assigned to different priority levels cannot starve each other. Within a priority level, a fair-queuing algorithm prevents requests from different flows from starving each other, and allows for requests to be queued to prevent bursty traffic from causing failed requests when the average load is acceptably low.

Priority Levels

Without APF enabled, overall concurrency in the API server is limited by the kube-apiserver flags --max-requests-inflight and --max-mutating-requests-inflight. With APF enabled, the concurrency limits defined by these flags are summed and then the sum is divided up among a configurable set of priority levels. Each incoming request is assigned to a single priority level, and each priority level will only dispatch as many concurrent requests as its configuration allows.

The default configuration, for example, includes separate priority levels for leader-election requests, requests from built-in controllers, and requests from Pods. This means that an ill-behaved Pod that floods the API server with requests cannot prevent leader election or actions by the built-in controllers from succeeding.

Queuing

Even within a priority level there may be a large number of distinct sources of traffic. In an overload situation, it is valuable to prevent one stream of requests from starving others (in particular, in the relatively common case of a single buggy client flooding the kube-apiserver with requests, that buggy client would ideally not have much measurable impact on other clients at all). This is handled by use of a fair-queuing algorithm to process requests that are assigned the same priority level. Each request is assigned to a flow, identified by the name of the matching FlowSchema plus a flow distinguisher — which is either the requesting user, the target resource's namespace, or nothing — and the system attempts to give approximately equal weight to requests in different flows of the same priority level.

After classifying a request into a flow, the API Priority and Fairness feature then may assign the request to a queue. This assignment uses a technique known as [shuffle sharding](#), which makes relatively efficient use of queues to insulate low-intensity flows from high-intensity flows.

The details of the queuing algorithm are tunable for each priority level, and allow administrators to trade off memory use, fairness (the property that independent flows will all make progress when total traffic exceeds capacity), tolerance for bursty traffic, and the added latency induced by queuing.

Exempt requests

Some requests are considered sufficiently important that they are not subject to any of the limitations imposed by this feature. These exemptions prevent an improperly-configured flow control configuration from totally disabling an API server.

Defaults

The Priority and Fairness feature ships with a suggested configuration that should suffice for experimentation; if your cluster is likely to experience heavy load then you should consider what configuration will work best. The suggested configuration groups requests into five priority classes:

- *The system priority level is for requests from the system:nodes group, i.e. Kubelets, which must be able to contact the API server in order for workloads to be able to schedule on them.*
- *The leader-election priority level is for leader election requests from built-in controllers (in particular, requests for endpoints, configmaps, or leases coming from the system:kube-controller-manager or system:kube-scheduler users and service accounts in the kube-system namespace). These are important to isolate from other traffic because failures in leader election cause their controllers to fail and restart, which in turn causes more expensive traffic as the new controllers sync their informers.*

- *The workload-high priority level is for other requests from built-in controllers.*
- *The workload-low priority level is for requests from any other service account, which will typically include all requests from controllers running in Pods.*
- *The global-default priority level handles all other traffic, e.g. interactive kubectl commands run by nonprivileged users.*

Additionally, there are two PriorityLevelConfigurations and two FlowSchemas that are built in and may not be overwritten:

- *The special exempt priority level is used for requests that are not subject to flow control at all: they will always be dispatched immediately. The special exempt FlowSchema classifies all requests from the system:masters group into this priority level. You may define other FlowSchemas that direct other requests to this priority level, if appropriate.*
- *The special catch-all priority level is used in combination with the special catch-all FlowSchema to make sure that every request gets some kind of classification. Typically you should not rely on this catch-all configuration, and should create your own catch-all FlowSchema and PriorityLevelConfiguration (or use the global-default configuration that is installed by default) as appropriate. To help catch configuration errors that miss classifying some requests, the mandatory catch-all priority level only allows one concurrency share and does not queue requests, making it relatively likely that traffic that only matches the catch-all FlowSchema will be rejected with an HTTP 429 error.*

Health check concurrency exemption

The suggested configuration gives no special treatment to the health check requests on kube-apiservers from their local kubelets --- which tend to use the secured port but supply no credentials. With the suggested config, these requests get assigned to the global-default FlowSchema and the corresponding global-default priority level, where other traffic can crowd them out.

If you add the following additional FlowSchema, this exempts those requests from rate limiting.

Caution: *Making this change also allows any hostile party to then send health-check requests that match this FlowSchema, at any volume they like. If you have a web traffic filter or similar external security mechanism to protect your cluster's API server from general internet traffic, you can configure rules to block any health check requests that originate from outside your cluster.*



```
apiVersion: flowcontrol.apiserver.k8s.io/v1beta1
kind: FlowSchema
metadata:
  name: health-for-strangers
spec:
  matchingPrecedence: 1000
  priorityLevelConfiguration:
    name: exempt
  rules:
    - nonResourceRules:
        - nonResourceURLs:
            - "/healthz"
            - "/livez"
            - "/readyz"
        verbs:
          - "*"
    subjects:
      - kind: Group
        group:
          name: system:unauthenticated
```

Resources

The flow control API involves two kinds of resources.

[*PriorityLevelConfigurations*](#) define the available isolation classes, the share of the available concurrency budget that each can handle, and allow for fine-tuning queuing behavior. [*FlowSchemas*](#) are used to classify individual inbound requests, matching each to a single *PriorityLevelConfiguration*. There is also a *v1alpha1* version of the same API group, and it has the same Kinds with the same syntax and semantics.

PriorityLevelConfiguration

A *PriorityLevelConfiguration* represents a single isolation class. Each *PriorityLevelConfiguration* has an independent limit on the number of outstanding requests, and limitations on the number of queued requests.

Concurrency limits for *PriorityLevelConfigurations* are not specified in absolute number of requests, but rather in "concurrency shares." The total concurrency limit for the API Server is distributed among the existing *PriorityLevelConfigurations* in proportion with these shares. This allows a cluster administrator to scale up or down the total amount of traffic to a server by restarting *kube-apiserver* with a different value for *--max-requests-inflight* (or *--max-mutating-requests-inflight*), and all *PriorityLevelConfigurations* will see their maximum allowed concurrency go up (or down) by the same fraction.

Caution: With the Priority and Fairness feature enabled, the total concurrency limit for the server is set to the sum of `--max-requests-inflight` and `--max-mutating-requests-inflight`. There is no longer any distinction made between mutating and non-mutating requests; if you want to treat them separately for a given resource, make separate FlowSchemas that match the mutating and non-mutating verbs respectively.

When the volume of inbound requests assigned to a single `PriorityLevelConfiguration` is more than its permitted concurrency level, the `type` field of its specification determines what will happen to extra requests. A type of `Reject` means that excess traffic will immediately be rejected with an HTTP 429 (Too Many Requests) error. A type of `Queue` means that requests above the threshold will be queued, with the shuffle sharding and fair queuing techniques used to balance progress between request flows.

The queuing configuration allows tuning the fair queuing algorithm for a priority level. Details of the algorithm can be read in the [enhancement proposal](#), but in short:

- Increasing `queues` reduces the rate of collisions between different flows, at the cost of increased memory usage. A value of 1 here effectively disables the fair-queuing logic, but still allows requests to be queued.
- Increasing `queueLengthLimit` allows larger bursts of traffic to be sustained without dropping any requests, at the cost of increased latency and memory usage.
- Changing `handSize` allows you to adjust the probability of collisions between different flows and the overall concurrency available to a single flow in an overload situation.

Note: A larger `handSize` makes it less likely for two individual flows to collide (and therefore for one to be able to starve the other), but more likely that a small number of flows can dominate the apiserver. A larger `handSize` also potentially increases the amount of latency that a single high-traffic flow can cause. The maximum number of queued requests possible from a single flow is `handSize * queueLengthLimit`.

Following is a table showing an interesting collection of shuffle sharding configurations, showing for each the probability that a given mouse (low-intensity flow) is squished by the elephants (high-intensity flows) for an illustrative collection of numbers of elephants. See <https://play.golang.org/p/GiOPLgVHiUq>, which computes this table.

HandSize	Queues	1 elephant	4 elephants	16 elephants
12	32	4.428838398950118e-09	0.11431348830099144	0.9935089607
10	32	1.550093439632541e-08	0.0626479840223545	0.9753101519
10	64	6.601827268370426e-12	0.00045571320990370776	0.4999992915

HandSize	Queues	1 elephant	4 elephants	16 elephants
9	64	3.6310049976037345e-11	0.00045501212304112273	0.4282314876
8	64	2.25929199850899e-10	0.0004886697053040446	0.3593511468
8	128	6.994461389026097e-13	3.4055790161620863e-06	0.0274617313
7	128	1.0579122850901972e-11	6.960839379258192e-06	0.0240615738
7	256	7.597695465552631e-14	6.728547142019406e-08	0.0006709661
6	256	2.7134626662687968e-12	2.9516464018476436e-07	0.0008895654
6	512	4.116062922897309e-14	4.982983350480894e-09	2.2602576434
6	1024	6.337324016514285e-16	8.09060164312957e-11	4.5174080629

FlowSchema

A *FlowSchema* matches some inbound requests and assigns them to a priority level. Every inbound request is tested against every *FlowSchema* in turn, starting with those with numerically lowest --- which we take to be the logically highest --- *matchingPrecedence* and working onward. The first match wins.

Caution: Only the first matching *FlowSchema* for a given request matters. If multiple *FlowSchemas* match a single inbound request, it will be assigned based on the one with the highest *matchingPrecedence*. If multiple *FlowSchemas* with equal *matchingPrecedence* match the same request, the one with lexicographically smaller name will win, but it's better not to rely on this, and instead to ensure that no two *FlowSchemas* have the same *matchingPrecedence*.

A *FlowSchema* matches a given request if at least one of its *rules* matches. A rule matches if at least one of its *subjects* and at least one of its *resourceRules* or *nonResourceRules* (depending on whether the incoming request is for a resource or non-resource URL) matches the request.

For the *name* field in subjects, and the verbs, *apiGroups*, *resources*, *namespaces*, and *nonResourceURLs* fields of resource and non-resource rules, the wildcard * may be specified to match all values for the given field, effectively removing it from consideration.

A *FlowSchema*'s *distinguisherMethod.type* determines how requests matching that schema will be separated into flows. It may be either *ByUser*, in which case one requesting user will not be able to starve other users of capacity, or *ByNamespace*, in which case requests for resources in one namespace will not be able to starve requests for resources in other namespaces of capacity, or it may be blank (or *distinguisherMethod* may be omitted entirely), in which case all requests matched by this *FlowSchema* will be considered part of a single flow. The correct choice for a given *FlowSchema* depends on the resource and your particular environment.

Diagnostics

Every HTTP response from an API server with the priority and fairness feature enabled has two extra headers: `X-Kubernetes-PF-FlowSchema-UID` and `X-Kubernetes-PF-PriorityLevel-UID`, noting the flow schema that matched the request and the priority level to which it was assigned, respectively. The API objects' names are not included in these headers in case the requesting user does not have permission to view them, so when debugging you can use a command like

```
kubectl get flowschemas -o custom-columns="uid:  
{metadata.uid},name:{metadata.name}"  
kubectl get prioritylevelconfigurations -o custom-columns="uid:  
{metadata.uid},name:{metadata.name}"
```

to get a mapping of UIDs to names for both `FlowSchemas` and `PriorityLevelConfigurations`.

Observability

Metrics

Note: In versions of Kubernetes before v1.20, the labels `flow_sch ema` and `priority_level` were inconsistently named `flowSchema` and `priorityLevel`, respectively. If you're running Kubernetes versions v1.19 and earlier, you should refer to the documentation for your version.

When you enable the API Priority and Fairness feature, the `kube-apiserver` exports additional metrics. Monitoring these can help you determine whether your configuration is inappropriately throttling important traffic, or find poorly-behaved workloads that may be harming system health.

- `apiserver_flowcontrol_rejected_requests_total` is a counter vector (cumulative since server start) of requests that were rejected, broken down by the labels `flow_schema` (indicating the one that matched the request), `priority_level` (indicating the one to which the request was assigned), and `reason`. The `reason` label will be have one of the following values:
 - `queue-full`, indicating that too many requests were already queued,
 - `concurrency-limit`, indicating that the `PriorityLevelConfiguration` is configured to reject rather than queue excess requests, or
 - `time-out`, indicating that the request was still in the queue when its queuing time limit expired.
- `apiserver_flowcontrol_dispatched_requests_total` is a counter vector (cumulative since server start) of requests that began executing, broken down by the labels `flow_schema` (indicating the one that

matched the request) and priority_level (indicating the one to which the request was assigned).

- *apiserver_current_inqueue_requests* is a gauge vector of recent high water marks of the number of queued requests, grouped by a label named *request_kind* whose value is *mutating* or *readOnly*. These high water marks describe the largest number seen in the one second window most recently completed. These complement the older *apiserver_current_inflight_requests* gauge vector that holds the last window's high water mark of number of requests actively being served.
- *apiserver_flowcontrol_read_vs_write_request_count_samples* is a histogram vector of observations of the then-current number of requests, broken down by the labels *phase* (which takes on the values *waiting* and *executing*) and *request_kind* (which takes on the values *mutating* and *readOnly*). The observations are made periodically at a high rate.
- *apiserver_flowcontrol_read_vs_write_request_count_watermarks* is a histogram vector of high or low water marks of the number of requests broken down by the labels *phase* (which takes on the values *waiting* and *executing*) and *request_kind* (which takes on the values *mutating* and *readOnly*); the label *mark* takes on values *high* and *low*. The water marks are accumulated over windows bounded by the times when an observation was added to *apiserver_flowcontrol_read_vs_write_request_count_samples*. These water marks show the range of values that occurred between samples.
- *apiserver_flowcontrol_current_inqueue_requests* is a gauge vector holding the instantaneous number of queued (not executing) requests, broken down by the labels *priority_level* and *flow_schema*.
- *apiserver_flowcontrol_current_executing_requests* is a gauge vector holding the instantaneous number of executing (not waiting in a queue) requests, broken down by the labels *priority_level* and *flow_schema*.
- *apiserver_flowcontrol_priority_level_request_count_samples* is a histogram vector of observations of the then-current number of requests broken down by the labels *phase* (which takes on the values *waiting* and *executing*) and *priority_level*. Each histogram gets observations taken periodically, up through the last activity of the relevant sort. The observations are made at a high rate.
- *apiserver_flowcontrol_priority_level_request_count_watermarks* is a histogram vector of high or low water marks of the number of requests broken down by the labels *phase* (which takes on the values *waiting* and *executing*) and *priority_level*; the label *mark* takes on values *high* and *low*. The water marks are accumulated over windows bounded by the times when an observation was added to *apiserver_flowcontrol_priority_level_request_count_samples*. These water marks show the range of values that occurred between samples.

- *apiserver_flowcontrol_request_queue_length_after_enqueue* is a histogram vector of queue lengths for the queues, broken down by the labels *priority_level* and *flow_schema*, as sampled by the enqueued requests. Each request that gets queued contributes one sample to its histogram, reporting the length of the queue just after the request was added. Note that this produces different statistics than an unbiased survey would.

Note: An outlier value in a histogram here means it is likely that a single flow (i.e., requests by one user or for one namespace, depending on configuration) is flooding the API server, and being throttled. By contrast, if one priority level's histogram shows that all queues for that priority level are longer than those for other priority levels, it may be appropriate to increase that PriorityLevelConfiguration's concurrency shares.

- *apiserver_flowcontrol_request_concurrency_limit* is a gauge vector holding the computed concurrency limit (based on the API server's total concurrency limit and PriorityLevelConfigurations' concurrency shares), broken down by the label *priority_level*.
- *apiserver_flowcontrol_request_wait_duration_seconds* is a histogram vector of how long requests spent queued, broken down by the labels *flow_schema* (indicating which one matched the request), *priority_level* (indicating the one to which the request was assigned), and *execute* (indicating whether the request started executing).

Note: Since each FlowSchema always assigns requests to a single PriorityLevelConfiguration, you can add the histograms for all the FlowSchemas for one priority level to get the effective histogram for requests assigned to that priority level.

- *apiserver_flowcontrol_request_execution_seconds* is a histogram vector of how long requests took to actually execute, broken down by the labels *flow_schema* (indicating which one matched the request) and *priority_level* (indicating the one to which the request was assigned).

Debug endpoints

When you enable the API Priority and Fairness feature, the kube-apiserver serves the following additional paths at its HTTP[S] ports.

- */debug/api_priority_and_fairness/dump_priority_levels* - a listing of all the priority levels and the current state of each. You can fetch like this:

```
kubectl get --raw /debug/api_priority_and_fairness/
dump_priority_levels
```

The output is similar to this:

```

PriorityLevelName, ActiveQueues, IsIdle, IsQuiescing,
WaitingRequests, ExecutingRequests,
workload-low,          0,           true,   false,
0,                   0,           true,   false,
global-default,        0,           true,   false,
0,                   0,           true,   false,
exempt,                <none>,     <none>, <none>,
<none>,                <none>,
catch-all,              0,           true,   false,
0,                   0,           true,   false,
system,                0,           true,   false,
0,                   0,           true,   false,
leader-election,       0,           true,   false,
0,                   0,           true,   false,
workload-high,         0,           true,   false,
0,                   0,

```

- `/debug/api_priority_and_fairness/dump_queues` - a listing of all the queues and their current state. You can fetch like this:

```
kubectl get --raw /debug/api_priority_and_fairness/
dump_queues
```

The output is similar to this:

```

PriorityLevelName, Index, PendingRequests,
ExecutingRequests, VirtualStart,
workload-high,      0,      0,
0,                 0.0000,
workload-high,      1,      0,
0,                 0.0000,
workload-high,      2,      0,
0,                 0.0000,
...
leader-election,    14,     0,
0,                 0.0000,
leader-election,    15,     0,
0,                 0.0000,

```

- `/debug/api_priority_and_fairness/dump_requests` - a listing of all the requests that are currently waiting in a queue. You can fetch like this:

```
kubectl get --raw /debug/api_priority_and_fairness/
dump_requests
```

The output is similar to this:

```

PriorityLevelName, FlowSchemaName, QueueIndex,
RequestIndexInQueue, FlowDistingsher,      ArriveTime,
exempt,            <none>,      <none>,
<none>,            <none>,      <none>,
system,            system-nodes, 12,

```

```
0, system:node:127.0.0.1,  
2020-07-23T15:26:57.179170694Z,
```

In addition to the queued requests, the output includes one phantom line for each priority level that is exempt from limitation.

You can get a more detailed listing with a command like this:

```
kubectl get --raw '/debug/api_priority_and_fairness/  
dump_requests?includeRequestDetails=1'
```

The output is similar to this:

```
PriorityLevelName, FlowSchemaName, QueueIndex,  
RequestIndexInQueue, FlowDistingsher,  
ArriveTime, UserName,  
Verb,  
APIPath,  
Namespace, Name, APIVersion, Resource, SubResource,  
system, system-nodes, 12,  
0, system:node:127.0.0.1,  
2020-07-23T15:31:03.583823404Z, system:node:127.0.0.1,  
create, /api/v1/namespaces/scaletest/configmaps,  
system, system-nodes, 12,  
1, system:node:127.0.0.1,  
2020-07-23T15:31:03.594555947Z, system:node:127.0.0.1,  
create, /api/v1/namespaces/scaletest/configmaps,
```

What's next

For background information on design details for API priority and fairness, see the [enhancement proposal](#). You can make suggestions and feature requests via [SIG API Machinery](#).

Feedback

Was this page helpful?

Yes No

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

Last modified November 21, 2020 at 10:53 PM PST: [Fix typo \(c0eafeb53\)](#)
[Edit this page](#) [Create child page](#) [Create an issue](#)

- [Enabling/Disabling API Priority and Fairness](#)
- [Concepts](#)
 - [Priority Levels](#)
 - [Queuing](#)
 - [Exempt requests](#)

- [Defaults](#)
- [Health check concurrency exemption](#)
- [Resources](#)
 - [PriorityLevelConfiguration](#)
 - [FlowSchema](#)
- [Diagnostics](#)
- [Observability](#)
 - [Metrics](#)
 - [Debug endpoints](#)
- [What's next](#)

Installing Addons

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

Add-ons extend the functionality of Kubernetes.

This page lists some of the available add-ons and links to their respective installation instructions.

Networking and Network Policy

- [ACI](#) provides integrated container networking and network security with Cisco ACI.
- [Calico](#) is a networking and network policy provider. Calico supports a flexible set of networking options so you can choose the most efficient option for your situation, including non-overlay and overlay networks, with or without BGP. Calico uses the same engine to enforce network policy for hosts, pods, and (if using Istio & Envoy) applications at the service mesh layer.
- [Canal](#) unites Flannel and Calico, providing networking and network policy.
- [Cilium](#) is a L3 network and network policy plugin that can enforce HTTP/API/L7 policies transparently. Both routing and overlay/encapsulation mode are supported, and it can work on top of other CNI plugins.
- [CNI-Genie](#) enables Kubernetes to seamlessly connect to a choice of CNI plugins, such as Calico, Canal, Flannel, Romana, or Weave.
- [Contiv](#) provides configurable networking (native L3 using BGP, overlay using vxlan, classic L2, and Cisco-SDN/ACI) for various use cases and a rich policy framework. Contiv project is fully [open sourced](#). The [installer](#) provides both kubeadm and non-kubeadm based installation options.
- [Contrail](#), based on [Tungsten Fabric](#), is an open source, multi-cloud network virtualization and policy management platform. Contrail and

Tungsten Fabric are integrated with orchestration systems such as Kubernetes, OpenShift, OpenStack and Mesos, and provide isolation modes for virtual machines, containers/pods and bare metal workloads.

- [Flannel](#) is an overlay network provider that can be used with Kubernetes.
- [Knitter](#) is a plugin to support multiple network interfaces in a Kubernetes pod.
- [Multus](#) is a Multi plugin for multiple network support in Kubernetes to support all CNI plugins (e.g. Calico, Cilium, Contiv, Flannel), in addition to SRIOV, DPDK, OVS-DPDK and VPP based workloads in Kubernetes.
- [OVN-Kubernetes](#) is a networking provider for Kubernetes based on [OVN \(Open Virtual Network\)](#), a virtual networking implementation that came out of the Open vSwitch (OVS) project. OVN-Kubernetes provides an overlay based networking implementation for Kubernetes, including an OVS based implementation of load balancing and network policy.
- [OVN4NFV-K8S-Plugin](#) is OVN based CNI controller plugin to provide cloud native based Service function chaining(SFC), Multiple OVN overlay networking, dynamic subnet creation, dynamic creation of virtual networks, VLAN Provider network, Direct provider network and pluggable with other Multi-network plugins, ideal for edge based cloud native workloads in Multi-cluster networking
- [NSX-T](#) Container Plug-in (NCP) provides integration between VMware NSX-T and container orchestrators such as Kubernetes, as well as integration between NSX-T and container-based CaaS/PaaS platforms such as Pivotal Container Service (PKS) and OpenShift.
- [Nuage](#) is an SDN platform that provides policy-based networking between Kubernetes Pods and non-Kubernetes environments with visibility and security monitoring.
- [Romana](#) is a Layer 3 networking solution for pod networks that also supports the [NetworkPolicy API](#). Kubeadm add-on installation details available [here](#).
- [Weave Net](#) provides networking and network policy, will carry on working on both sides of a network partition, and does not require an external database.

Service Discovery

- [CoreDNS](#) is a flexible, extensible DNS server which can be [installed](#) as the in-cluster DNS for pods.

Visualization & Control

- [Dashboard](#) is a dashboard web interface for Kubernetes.
- [Weave Scope](#) is a tool for graphically visualizing your containers, pods, services etc. Use it in conjunction with a [Weave Cloud account](#) or host the UI yourself.

Infrastructure

- [KubeVirt](#) is an add-on to run virtual machines on Kubernetes. Usually run on bare-metal clusters.

Legacy Add-ons

There are several other add-ons documented in the deprecated [cluster/addons](#) directory.

Well-maintained ones should be linked to here. PRs welcome!

Feedback

Was this page helpful?

Yes No

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

Last modified November 04, 2020 at 5:09 PM PST: [Add OVN-Kubernetes to list of cluster add-ons \(21b52415e\)](#)

[Edit this page](#) [Create child page](#) [Create an issue](#)

- [Networking and Network Policy](#)
- [Service Discovery](#)
- [Visualization & Control](#)
- [Infrastructure](#)
- [Legacy Add-ons](#)

Extending Kubernetes

Different ways to change the behavior of your Kubernetes cluster.

Kubernetes is highly configurable and extensible. As a result, there is rarely a need to fork or submit patches to the Kubernetes project code.

This guide describes the options for customizing a Kubernetes cluster. It is aimed at [cluster operators](#) who want to understand how to adapt their Kubernetes cluster to the needs of their work environment. Developers who are prospective [Platform Developers](#) or Kubernetes Project [Contributors](#) will also find it useful as an introduction to what extension points and patterns exist, and their trade-offs and limitations.

Overview

Customization approaches can be broadly divided into configuration, which only involves changing flags, local configuration files, or API resources; and extensions, which involve running additional programs or services. This document is primarily about extensions.

Configuration

Configuration files and flags are documented in the Reference section of the online documentation, under each binary:

- [*kubelet*](#)
- [*kube-apiserver*](#)
- [*kube-controller-manager*](#)
- [*kube-scheduler*](#).

Flags and configuration files may not always be changeable in a hosted Kubernetes service or a distribution with managed installation. When they are changeable, they are usually only changeable by the cluster administrator. Also, they are subject to change in future Kubernetes versions, and setting them may require restarting processes. For those reasons, they should be used only when there are no other options.

*Built-in Policy APIs, such as [*ResourceQuota*](#), [*PodSecurityPolicies*](#), [*NetworkPolicy*](#) and Role-based Access Control ([*RBAC*](#)), are built-in Kubernetes APIs. APIs are typically used with hosted Kubernetes services and with managed Kubernetes installations. They are declarative and use the same conventions as other Kubernetes resources like pods, so new cluster configuration can be repeatable and be managed the same way as applications. And, where they are stable, they enjoy a [*defined support policy*](#) like other Kubernetes APIs. For these reasons, they are preferred over configuration files and flags where suitable.*

Extensions

Extensions are software components that extend and deeply integrate with Kubernetes. They adapt it to support new types and new kinds of hardware.

Most cluster administrators will use a hosted or distribution instance of Kubernetes. As a result, most Kubernetes users will not need to install extensions and fewer will need to author new ones.

Extension Patterns

Kubernetes is designed to be automated by writing client programs. Any program that reads and/or writes to the Kubernetes API can provide useful automation. Automation can run on the cluster or off it. By following the guidance in this doc you can write highly available and robust automation.

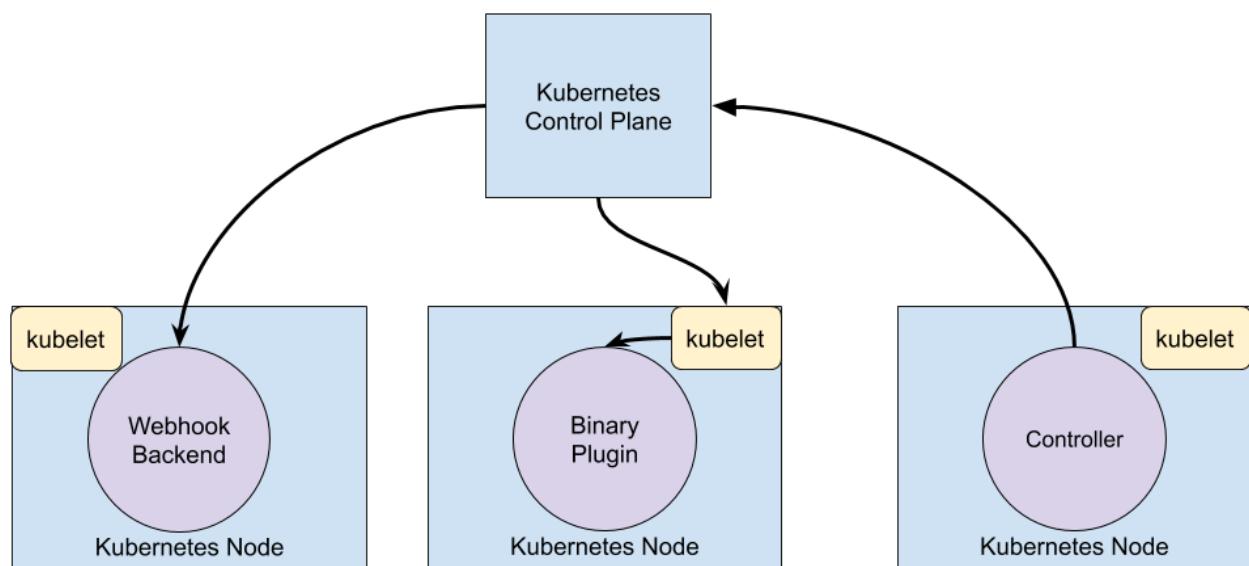
Automation generally works with any Kubernetes cluster, including hosted clusters and managed installations.

There is a specific pattern for writing client programs that work well with Kubernetes called the Controller pattern. Controllers typically read an object's `.spec`, possibly do things, and then update the object's `.status`.

A controller is a client of Kubernetes. When Kubernetes is the client and calls out to a remote service, it is called a Webhook. The remote service is called a Webhook Backend. Like Controllers, Webhooks do add a point of failure.

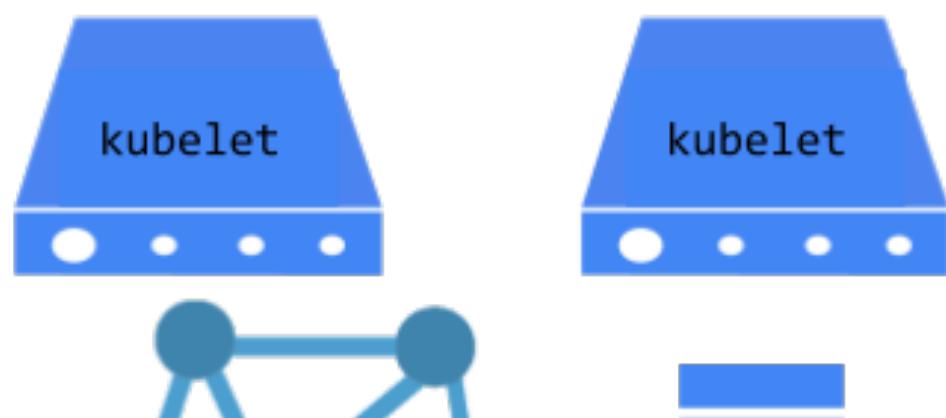
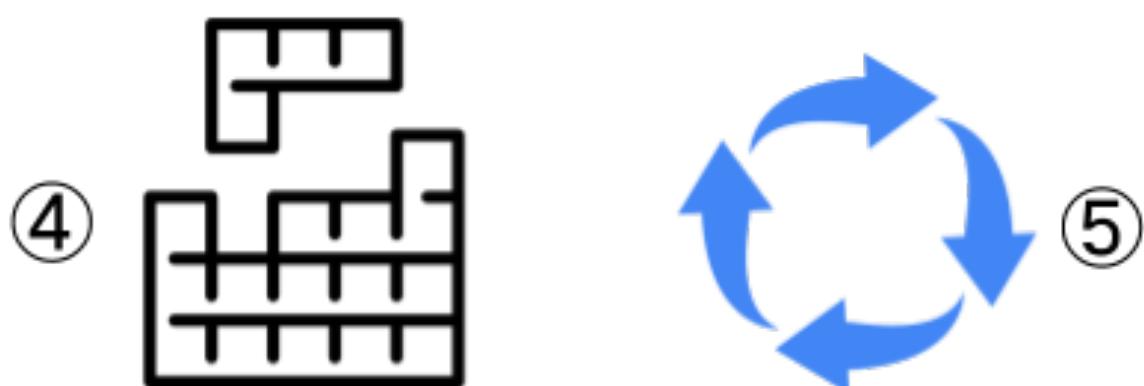
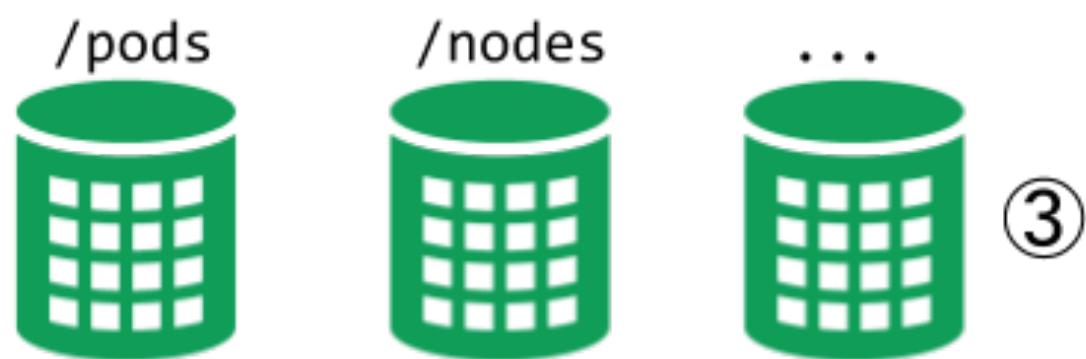
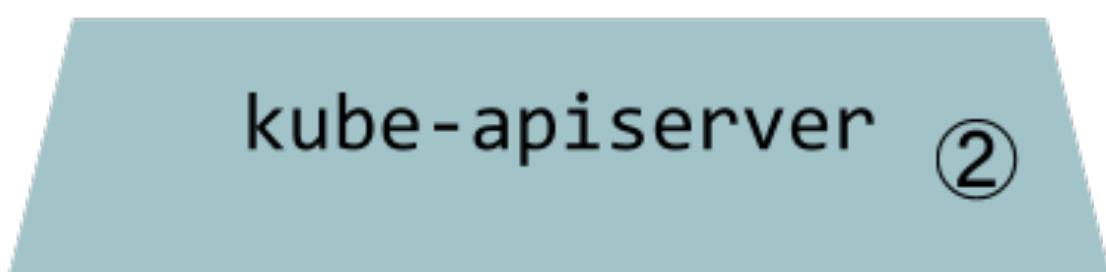
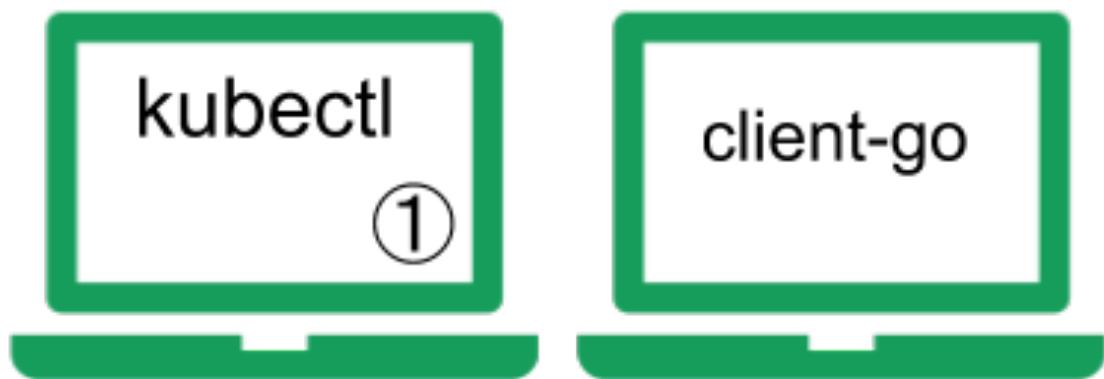
In the webhook model, Kubernetes makes a network request to a remote service. In the Binary Plugin model, Kubernetes executes a binary (program). Binary plugins are used by the kubelet (e.g. [Flex Volume Plugins](#) and [Network Plugins](#)) and by kubectl.

Below is a diagram showing how the extension points interact with the Kubernetes control plane.



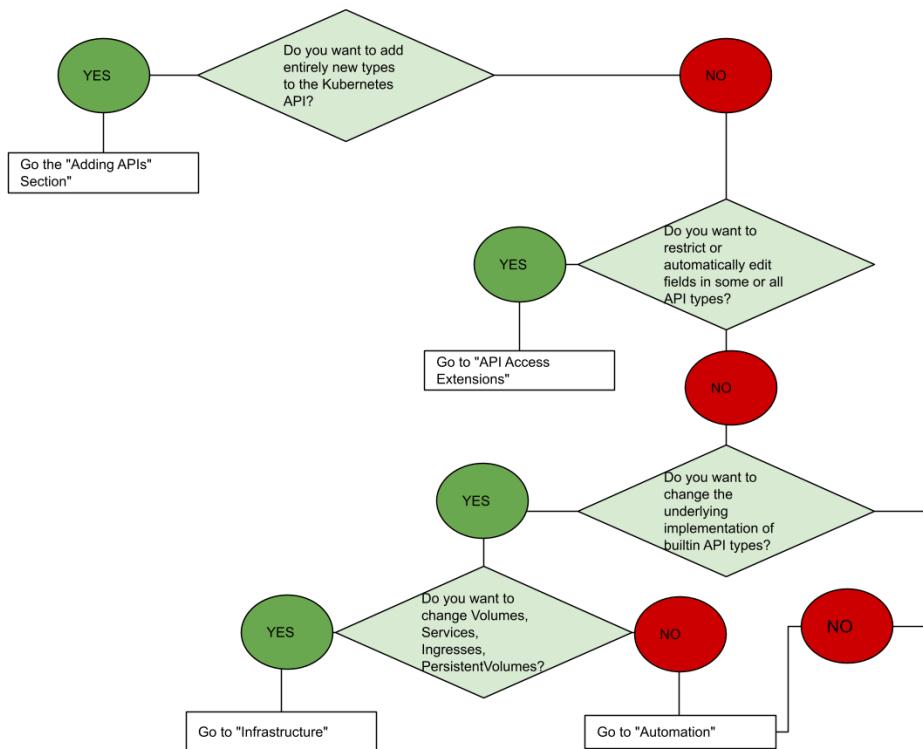
Extension Points

This diagram shows the extension points in a Kubernetes system.



1. Users often interact with the Kubernetes API using `kubectl`. [Kubectl plugins](#) extend the `kubectl` binary. They only affect the individual user's local environment, and so cannot enforce site-wide policies.
2. The apiserver handles all requests. Several types of extension points in the apiserver allow authenticating requests, or blocking them based on their content, editing content, and handling deletion. These are described in the [API Access Extensions](#) section.
3. The apiserver serves various kinds of resources. Built-in resource kinds, like `pods`, are defined by the Kubernetes project and can't be changed. You can also add resources that you define, or that other projects have defined, called Custom Resources, as explained in the [Custom Resources](#) section. Custom Resources are often used with API Access Extensions.
4. The Kubernetes scheduler decides which nodes to place pods on. There are several ways to extend scheduling. These are described in the [Scheduler Extensions](#) section.
5. Much of the behavior of Kubernetes is implemented by programs called Controllers which are clients of the API-Server. Controllers are often used in conjunction with Custom Resources.
6. The kubelet runs on servers, and helps pods appear like virtual servers with their own IPs on the cluster network. [Network Plugins](#) allow for different implementations of pod networking.
7. The kubelet also mounts and unmounts volumes for containers. New types of storage can be supported via [Storage Plugins](#).

If you are unsure where to start, this flowchart can help. Note that some solutions may involve several types of extensions.



API Extensions

User-Defined Types

Consider adding a Custom Resource to Kubernetes if you want to define new controllers, application configuration objects or other declarative APIs, and to manage them using Kubernetes tools, such as kubectl.

Do not use a Custom Resource as data storage for application, user, or monitoring data.

For more about Custom Resources, see the [Custom Resources concept guide](#).

Combining New APIs with Automation

The combination of a custom resource API and a control loop is called the [Operator pattern](#). The Operator pattern is used to manage specific, usually

stateful, applications. These custom APIs and control loops can also be used to control other resources, such as storage or policies.

Changing Built-in Resources

When you extend the Kubernetes API by adding custom resources, the added resources always fall into a new API Groups. You cannot replace or change existing API groups. Adding an API does not directly let you affect the behavior of existing APIs (e.g. Pods), but API Access Extensions do.

API Access Extensions

When a request reaches the Kubernetes API Server, it is first Authenticated, then Authorized, then subject to various types of Admission Control. See [Controlling Access to the Kubernetes API](#) for more on this flow.

Each of these steps offers extension points.

Kubernetes has several built-in authentication methods that it supports. It can also sit behind an authenticating proxy, and it can send a token from an Authorization header to a remote service for verification (a webhook). All of these methods are covered in the [Authentication documentation](#).

Authentication

[Authentication](#) maps headers or certificates in all requests to a username for the client making the request.

Kubernetes provides several built-in authentication methods, and an [Authentication webhook](#) method if those don't meet your needs.

Authorization

[Authorization](#) determines whether specific users can read, write, and do other operations on API resources. It just works at the level of whole resources -- it doesn't discriminate based on arbitrary object fields. If the built-in authorization options don't meet your needs, and [Authorization webhook](#) allows calling out to user-provided code to make an authorization decision.

Dynamic Admission Control

After a request is authorized, if it is a write operation, it also goes through [Admission Control](#) steps. In addition to the built-in steps, there are several extensions:

- The [Image Policy webhook](#) restricts what images can be run in containers.
- To make arbitrary admission control decisions, a general [Admission webhook](#) can be used. Admission Webhooks can reject creations or updates.

Infrastructure Extensions

Storage Plugins

[Flex Volumes](#) allow users to mount volume types without built-in support by having the Kubelet call a Binary Plugin to mount the volume.

Device Plugins

Device plugins allow a node to discover new Node resources (in addition to the builtin ones like cpu and memory) via a [Device Plugin](#).

Network Plugins

Different networking fabrics can be supported via node-level [Network Plugins](#).

Scheduler Extensions

The scheduler is a special type of controller that watches pods, and assigns pods to nodes. The default scheduler can be replaced entirely, while continuing to use other Kubernetes components, or [multiple schedulers](#) can run at the same time.

This is a significant undertaking, and almost all Kubernetes users find they do not need to modify the scheduler.

The scheduler also supports a [webhook](#) that permits a webhook backend (scheduler extension) to filter and prioritize the nodes chosen for a pod.

What's next

- Learn more about [Custom Resources](#)
- Learn about [Dynamic admission control](#)
- Learn more about Infrastructure extensions
 - [Network Plugins](#)
 - [Device Plugins](#)
- Learn about [kubectl plugins](#)
- Learn about the [Operator pattern](#)

Extending your Kubernetes Cluster

Kubernetes is highly configurable and extensible. As a result, there is rarely a need to fork or submit patches to the Kubernetes project code.

This guide describes the options for customizing a Kubernetes cluster. It is aimed at [cluster operators](#) who want to understand how to adapt their Kubernetes cluster to the needs of their work environment. Developers who are prospective [Platform Developers](#) or Kubernetes Project [Contributors](#) will also find it useful as an introduction to what extension points and patterns exist, and their trade-offs and limitations.

Overview

Customization approaches can be broadly divided into configuration, which only involves changing flags, local configuration files, or API resources; and extensions, which involve running additional programs or services. This document is primarily about extensions.

Configuration

Configuration files and flags are documented in the Reference section of the online documentation, under each binary:

- [kubelet](#)
- [kube-apiserver](#)
- [kube-controller-manager](#)
- [kube-scheduler](#).

Flags and configuration files may not always be changeable in a hosted Kubernetes service or a distribution with managed installation. When they

are changeable, they are usually only changeable by the cluster administrator. Also, they are subject to change in future Kubernetes versions, and setting them may require restarting processes. For those reasons, they should be used only when there are no other options.

Built-in Policy APIs, such as [ResourceQuota](#), [PodSecurityPolicies](#), [NetworkPolicy](#) and Role-based Access Control ([RBAC](#)), are built-in Kubernetes APIs. APIs are typically used with hosted Kubernetes services and with managed Kubernetes installations. They are declarative and use the same conventions as other Kubernetes resources like pods, so new cluster configuration can be repeatable and be managed the same way as applications. And, where they are stable, they enjoy a [defined support policy](#) like other Kubernetes APIs. For these reasons, they are preferred over configuration files and flags where suitable.

Extensions

Extensions are software components that extend and deeply integrate with Kubernetes. They adapt it to support new types and new kinds of hardware.

Most cluster administrators will use a hosted or distribution instance of Kubernetes. As a result, most Kubernetes users will not need to install extensions and fewer will need to author new ones.

Extension Patterns

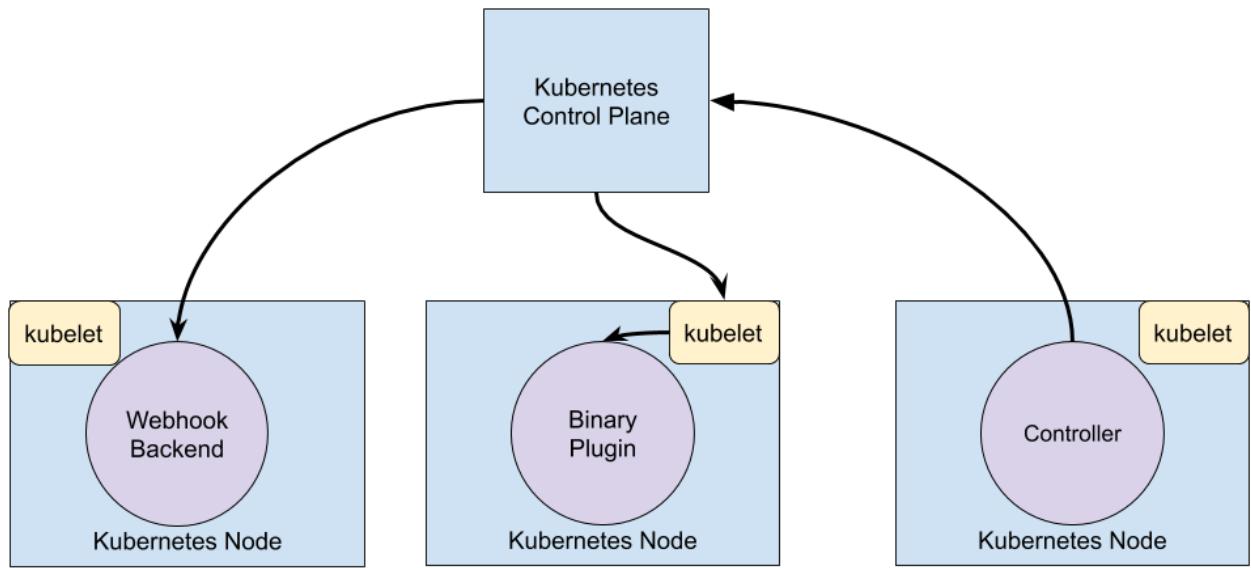
Kubernetes is designed to be automated by writing client programs. Any program that reads and/or writes to the Kubernetes API can provide useful automation. Automation can run on the cluster or off it. By following the guidance in this doc you can write highly available and robust automation. Automation generally works with any Kubernetes cluster, including hosted clusters and managed installations.

There is a specific pattern for writing client programs that work well with Kubernetes called the Controller pattern. Controllers typically read an object's `.spec`, possibly do things, and then update the object's `.status`.

A controller is a client of Kubernetes. When Kubernetes is the client and calls out to a remote service, it is called a Webhook. The remote service is called a Webhook Backend. Like Controllers, Webhooks do add a point of failure.

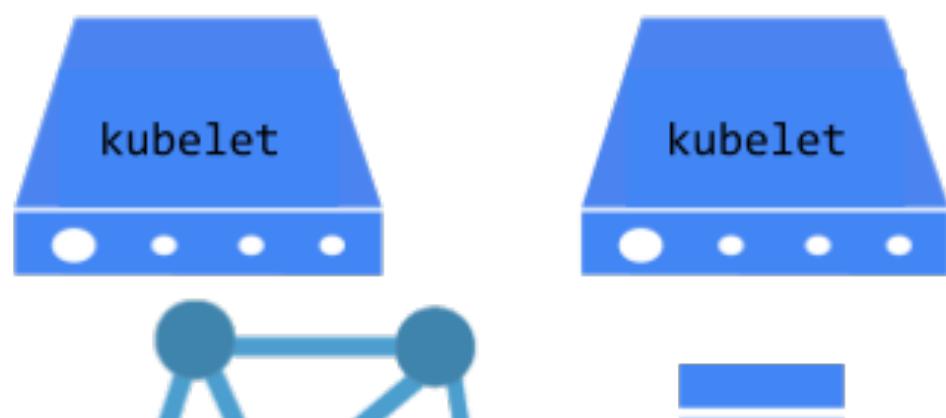
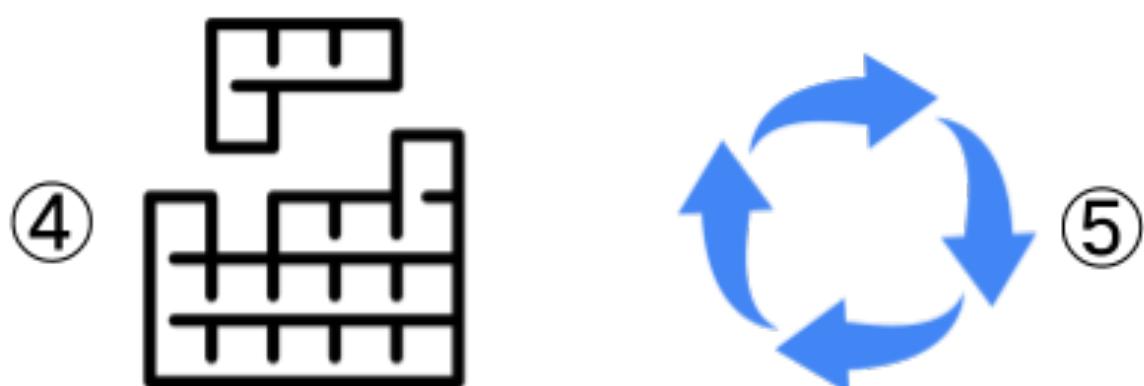
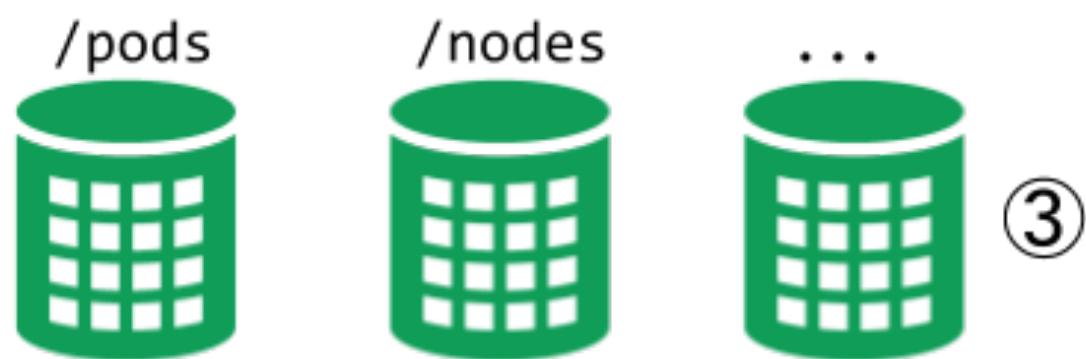
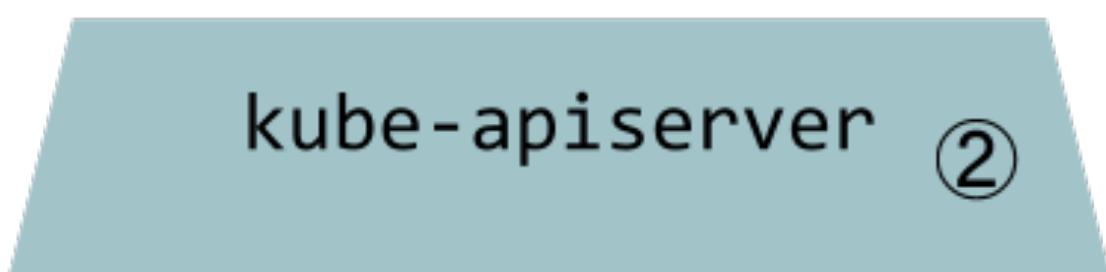
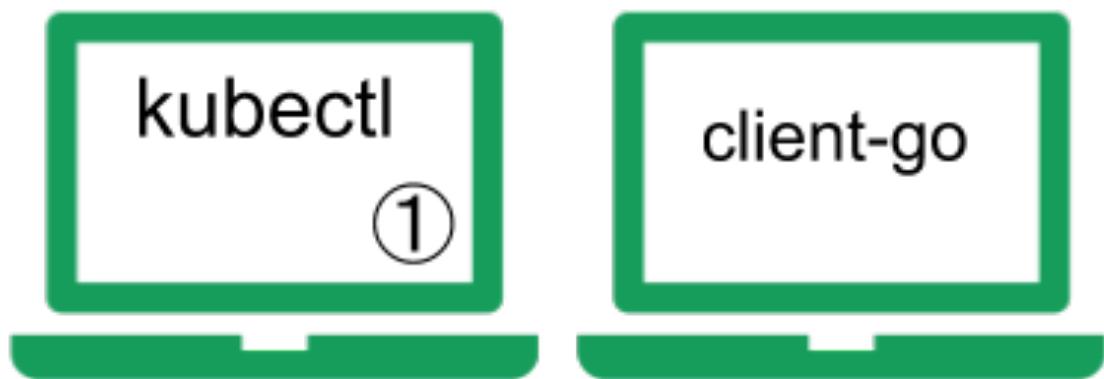
In the webhook model, Kubernetes makes a network request to a remote service. In the Binary Plugin model, Kubernetes executes a binary (program). Binary plugins are used by the kubelet (e.g. [Flex Volume Plugins](#) and [Network Plugins](#)) and by kubectl.

Below is a diagram showing how the extension points interact with the Kubernetes control plane.



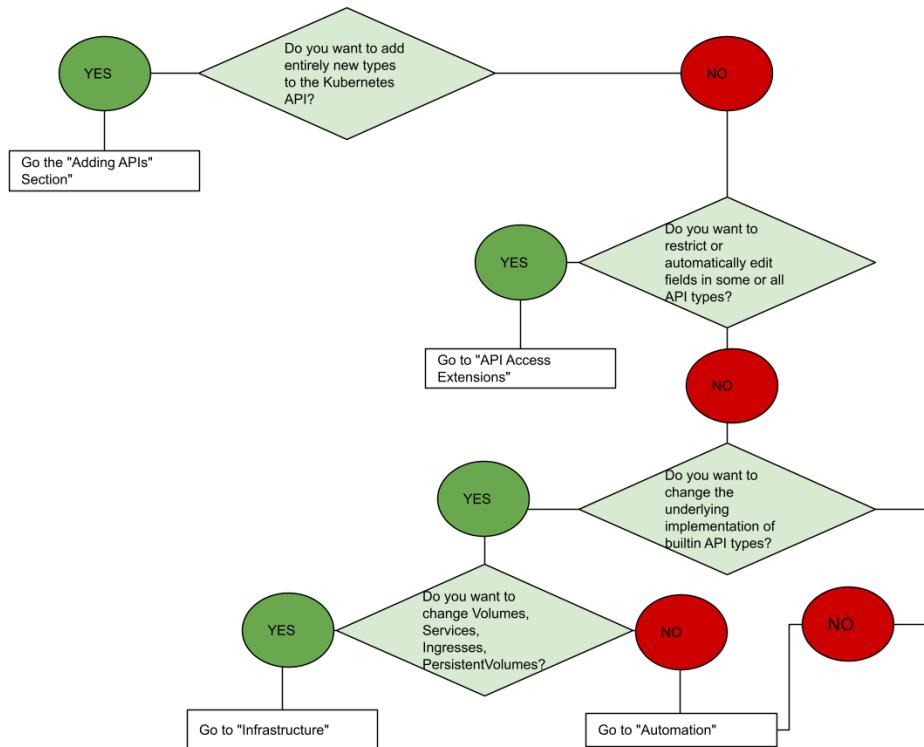
Extension Points

This diagram shows the extension points in a Kubernetes system.



1. Users often interact with the Kubernetes API using `kubectl`. [Kubectl plugins](#) extend the `kubectl` binary. They only affect the individual user's local environment, and so cannot enforce site-wide policies.
2. The apiserver handles all requests. Several types of extension points in the apiserver allow authenticating requests, or blocking them based on their content, editing content, and handling deletion. These are described in the [API Access Extensions](#) section.
3. The apiserver serves various kinds of resources. Built-in resource kinds, like `pods`, are defined by the Kubernetes project and can't be changed. You can also add resources that you define, or that other projects have defined, called Custom Resources, as explained in the [Custom Resources](#) section. Custom Resources are often used with API Access Extensions.
4. The Kubernetes scheduler decides which nodes to place pods on. There are several ways to extend scheduling. These are described in the [Scheduler Extensions](#) section.
5. Much of the behavior of Kubernetes is implemented by programs called Controllers which are clients of the API-Server. Controllers are often used in conjunction with Custom Resources.
6. The kubelet runs on servers, and helps pods appear like virtual servers with their own IPs on the cluster network. [Network Plugins](#) allow for different implementations of pod networking.
7. The kubelet also mounts and unmounts volumes for containers. New types of storage can be supported via [Storage Plugins](#).

If you are unsure where to start, this flowchart can help. Note that some solutions may involve several types of extensions.



API Extensions

User-Defined Types

Consider adding a Custom Resource to Kubernetes if you want to define new controllers, application configuration objects or other declarative APIs, and to manage them using Kubernetes tools, such as kubectl.

Do not use a Custom Resource as data storage for application, user, or monitoring data.

For more about Custom Resources, see the [Custom Resources concept guide](#).

Combining New APIs with Automation

The combination of a custom resource API and a control loop is called the [Operator pattern](#). The Operator pattern is used to manage specific, usually stateful, applications. These custom APIs and control loops can also be used to control other resources, such as storage or policies.

Changing Built-in Resources

When you extend the Kubernetes API by adding custom resources, the added resources always fall into a new API Groups. You cannot replace or change existing API groups. Adding an API does not directly let you affect the behavior of existing APIs (e.g. Pods), but API Access Extensions do.

API Access Extensions

When a request reaches the Kubernetes API Server, it is first Authenticated, then Authorized, then subject to various types of Admission Control. See [Controlling Access to the Kubernetes API](#) for more on this flow.

Each of these steps offers extension points.

Kubernetes has several built-in authentication methods that it supports. It can also sit behind an authenticating proxy, and it can send a token from an Authorization header to a remote service for verification (a webhook). All of these methods are covered in the [Authentication documentation](#).

Authentication

[Authentication](#) maps headers or certificates in all requests to a username for the client making the request.

Kubernetes provides several built-in authentication methods, and an [Authentication webhook](#) method if those don't meet your needs.

Authorization

[Authorization](#) determines whether specific users can read, write, and do other operations on API resources. It just works at the level of whole resources -- it doesn't discriminate based on arbitrary object fields. If the built-in authorization options don't meet your needs, and [Authorization webhook](#) allows calling out to user-provided code to make an authorization decision.

Dynamic Admission Control

After a request is authorized, if it is a write operation, it also goes through [Admission Control](#) steps. In addition to the built-in steps, there are several extensions:

- The [Image Policy webhook](#) restricts what images can be run in containers.
- To make arbitrary admission control decisions, a general [Admission webhook](#) can be used. Admission Webhooks can reject creations or updates.

Infrastructure Extensions

Storage Plugins

[Flex Volumes](#) allow users to mount volume types without built-in support by having the Kubelet call a Binary Plugin to mount the volume.

Device Plugins

Device plugins allow a node to discover new Node resources (in addition to the builtin ones like cpu and memory) via a [Device Plugin](#).

Network Plugins

Different networking fabrics can be supported via node-level [Network Plugins](#).

Scheduler Extensions

The scheduler is a special type of controller that watches pods, and assigns pods to nodes. The default scheduler can be replaced entirely, while continuing to use other Kubernetes components, or [multiple schedulers](#) can run at the same time.

This is a significant undertaking, and almost all Kubernetes users find they do not need to modify the scheduler.

The scheduler also supports a [webhook](#) that permits a webhook backend (scheduler extension) to filter and prioritize the nodes chosen for a pod.

What's next

- Learn more about [Custom Resources](#)
- Learn about [Dynamic admission control](#)
- Learn more about Infrastructure extensions
 - [Network Plugins](#)
 - [Device Plugins](#)
- Learn about [kubectl plugins](#)
- Learn about the [Operator pattern](#)

Feedback

Was this page helpful?

Yes No

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

Last modified October 13, 2020 at 12:41 AM PST: [Transfer "Controlling Access to the Kubernetes API" to the Concepts section \(78351ecaf\)](#)
[Edit this page](#) [Create child page](#) [Create an issue](#)

- [Overview](#)
- [Configuration](#)
- [Extensions](#)
- [Extension Patterns](#)
- [Extension Points](#)
- [API Extensions](#)
 - [User-Defined Types](#)
 - [Combining New APIs with Automation](#)
 - [Changing Built-in Resources](#)
 - [API Access Extensions](#)
 - [Authentication](#)
 - [Authorization](#)
 - [Dynamic Admission Control](#)
- [Infrastructure Extensions](#)
 - [Storage Plugins](#)
 - [Device Plugins](#)
 - [Network Plugins](#)
 - [Scheduler Extensions](#)
- [What's next](#)

Extending the Kubernetes API

[Custom Resources](#)

[Extending the Kubernetes API with the aggregation layer](#)

Custom Resources

Custom resources are extensions of the Kubernetes API. This page discusses when to add a custom resource to your Kubernetes cluster and when to use a standalone service. It describes the two methods for adding custom resources and how to choose between them.

Custom resources

A resource is an endpoint in the [Kubernetes API](#) that stores a collection of [API objects](#) of a certain kind; for example, the built-in pods resource contains a collection of Pod objects.

A custom resource is an extension of the Kubernetes API that is not necessarily available in a default Kubernetes installation. It represents a

customization of a particular Kubernetes installation. However, many core Kubernetes functions are now built using custom resources, making Kubernetes more modular.

Custom resources can appear and disappear in a running cluster through dynamic registration, and cluster admins can update custom resources independently of the cluster itself. Once a custom resource is installed, users can create and access its objects using [kubectl](#), just as they do for built-in resources like Pods.

Custom controllers

On their own, custom resources simply let you store and retrieve structured data. When you combine a custom resource with a custom controller, custom resources provide a true declarative API.

A [declarative API](#) allows you to declare or specify the desired state of your resource and tries to keep the current state of Kubernetes objects in sync with the desired state. The controller interprets the structured data as a record of the user's desired state, and continually maintains this state.

You can deploy and update a custom controller on a running cluster, independently of the cluster's lifecycle. Custom controllers can work with any kind of resource, but they are especially effective when combined with custom resources. The [Operator pattern](#) combines custom resources and custom controllers. You can use custom controllers to encode domain knowledge for specific applications into an extension of the Kubernetes API.

Should I add a custom resource to my Kubernetes Cluster?

When creating a new API, consider whether to [aggregate your API with the Kubernetes cluster APIs](#) or let your API stand alone.

Consider API aggregation if:	Prefer a stand-alone API if:
Your API is Declarative .	Your API does not fit the Declarative model.
You want your new types to be readable and writable using <code>kubectl</code> .	<code>kubectl</code> support is not required
You want to view your new types in a Kubernetes UI, such as <code>dashboard</code> , alongside built-in types.	Kubernetes UI support is not required.
You are developing a new API.	You already have a program that serves your API and works well.
You are willing to accept the format restriction that Kubernetes puts on REST resource paths, such as API Groups and Namespaces. (See the API Overview .)	You need to have specific REST paths to be compatible with an already defined REST API.

Consider API aggregation if:	Prefer a stand-alone API if:
Your resources are naturally scoped to a cluster or namespaces of a cluster.	Cluster or namespace scoped resources are a poor fit; you need control over the specifics of resource paths.
You want to reuse Kubernetes API support features .	You don't need those features.

Declarative APIs

In a Declarative API, typically:

- Your API consists of a relatively small number of relatively small objects (resources).
- The objects define configuration of applications or infrastructure.
- The objects are updated relatively infrequently.
- Humans often need to read and write the objects.
- The main operations on the objects are CRUD-y (creating, reading, updating and deleting).
- Transactions across objects are not required: the API represents a desired state, not an exact state.

Imperative APIs are not declarative. Signs that your API might not be declarative include:

- The client says "do this", and then gets a synchronous response back when it is done.
- The client says "do this", and then gets an operation ID back, and has to check a separate Operation object to determine completion of the request.
- You talk about Remote Procedure Calls (RPCs).
- Directly storing large amounts of data; for example, > a few kB per object, or > 1000s of objects.
- High bandwidth access (10s of requests per second sustained) needed.
- Store end-user data (such as images, PII, etc.) or other large-scale data processed by applications.
- The natural operations on the objects are not CRUD-y.
- The API is not easily modeled as objects.
- You chose to represent pending operations with an operation ID or an operation object.

Should I use a configMap or a custom resource?

Use a ConfigMap if any of the following apply:

- There is an existing, well-documented config file format, such as a `mysql.cnf` or `pom.xml`.
- You want to put the entire config file into one key of a configMap.

- The main use of the config file is for a program running in a Pod on your cluster to consume the file to configure itself.
- Consumers of the file prefer to consume via file in a Pod or environment variable in a pod, rather than the Kubernetes API.
- You want to perform rolling updates via Deployment, etc., when the file is updated.

Note: Use a [secret](#) for sensitive data, which is similar to a configMap but more secure.

Use a custom resource (CRD or Aggregated API) if most of the following apply:

- You want to use Kubernetes client libraries and CLIs to create and update the new resource.
- You want top-level support from kubectl; for example, `kubectl get my-object object-name`.
- You want to build new automation that watches for updates on the new object, and then CRUD other objects, or vice versa.
- You want to write automation that handles updates to the object.
- You want to use Kubernetes API conventions like `.spec`, `.status`, and `.metadata`.
- You want the object to be an abstraction over a collection of controlled resources, or a summarization of other resources.

Adding custom resources

Kubernetes provides two ways to add custom resources to your cluster:

- CRDs are simple and can be created without any programming.
- [API Aggregation](#) requires programming, but allows more control over API behaviors like how data is stored and conversion between API versions.

Kubernetes provides these two options to meet the needs of different users, so that neither ease of use nor flexibility is compromised.

Aggregated APIs are subordinate API servers that sit behind the primary API server, which acts as a proxy. This arrangement is called [API Aggregation](#) (AA). To users, it simply appears that the Kubernetes API is extended.

CRDs allow users to create new types of resources without adding another API server. You do not need to understand API Aggregation to use CRDs.

Regardless of how they are installed, the new resources are referred to as Custom Resources to distinguish them from built-in Kubernetes resources (like pods).

CustomResourceDefinitions

The [CustomResourceDefinition](#) API resource allows you to define custom resources. Defining a CRD object creates a new custom resource with a

name and schema that you specify. The Kubernetes API serves and handles the storage of your custom resource. The name of a CRD object must be a valid [DNS subdomain name](#).

This frees you from writing your own API server to handle the custom resource, but the generic nature of the implementation means you have less flexibility than with [API server aggregation](#).

Refer to the [custom controller example](#) for an example of how to register a new custom resource, work with instances of your new resource type, and use a controller to handle events.

API server aggregation

Usually, each resource in the Kubernetes API requires code that handles REST requests and manages persistent storage of objects. The main Kubernetes API server handles built-in resources like pods and services, and can also generically handle custom resources through [CRDs](#).

The [aggregation layer](#) allows you to provide specialized implementations for your custom resources by writing and deploying your own standalone API server. The main API server delegates requests to you for the custom resources that you handle, making them available to all of its clients.

Choosing a method for adding custom resources

CRDs are easier to use. Aggregated APIs are more flexible. Choose the method that best meets your needs.

Typically, CRDs are a good fit if:

- You have a handful of fields
- You are using the resource within your company, or as part of a small open-source project (as opposed to a commercial product)

Comparing ease of use

CRDs are easier to create than Aggregated APIs.

CRDs	Aggregated API
<i>Do not require programming. Users can choose any language for a CRD controller.</i>	<i>Requires programming in Go and building binary and image.</i>
<i>No additional service to run; CRDs are handled by API server.</i>	<i>An additional service to create and that could fail.</i>
<i>No ongoing support once the CRD is created. Any bug fixes are picked up as part of normal Kubernetes Master upgrades.</i>	<i>May need to periodically pickup bug fixes from upstream and rebuild and update the Aggregated API server.</i>

CRDs	Aggregated API
<i>No need to handle multiple versions of your API; for example, when you control the client for this resource, you can upgrade it in sync with the API.</i>	<i>You need to handle multiple versions of your API; for example, when developing an extension to share with the world.</i>

Advanced features and flexibility

Aggregated APIs offer more advanced API features and customization of other features; for example, the storage layer.

Feature	Description	CRDs	Aggregated API
Validation	<i>Help users prevent errors and allow you to evolve your API independently of your clients. These features are most useful when there are many clients who can't all update at the same time.</i>	<i>Yes. Most validation can be specified in the CRD using OpenAPI v3.0 validation. Any other validations supported by addition of a Validating Webhook.</i>	<i>Yes, arbitrary validation checks</i>
Defaulting	See above	<i>Yes, either via OpenAPI v3.0 validation default keyword (GA in 1.17), or via a Mutating Webhook (though this will not be run when reading from etcd for old objects).</i>	Yes
Multi-versioning	<i>Allows serving the same object through two API versions. Can help ease API changes like renaming fields. Less important if you control your client versions.</i>	Yes	Yes
Custom Storage	<i>If you need storage with a different performance mode (for example, a time-series database instead of key-value store) or isolation for security (for example, encryption of sensitive information, etc.)</i>	No	Yes

Feature	Description	CRDs	Aggregated API
<i>Custom Business Logic</i>	<i>Perform arbitrary checks or actions when creating, reading, updating or deleting an object</i>	Yes, using Webhooks .	Yes
<i>Scale Subresource</i>	<i>Allows systems like HorizontalPodAutoscaler and PodDisruptionBudget interact with your new resource</i>	Yes	Yes
<i>Status Subresource</i>	<i>Allows fine-grained access control where user writes the spec section and the controller writes the status section. Allows incrementing object Generation on custom resource data mutation (requires separate spec and status sections in the resource)</i>	Yes	Yes
<i>Other Subresources</i>	<i>Add operations other than CRUD, such as "logs" or "exec".</i>	No	Yes
<i>strategic-merge-patch</i>	<i>The new endpoints support PATCH with Content-Type: application/strategic-merge-patch+json. Useful for updating objects that may be modified both locally, and by the server. For more information, see "Update API Objects in Place Using kubectl patch"</i>	No	Yes
<i>Protocol Buffers</i>	<i>The new resource supports clients that want to use Protocol Buffers</i>	No	Yes
<i>OpenAPI Schema</i>	<i>Is there an OpenAPI (swagger) schema for the types that can be dynamically fetched from the server? Is the user protected from misspelling field names by ensuring only allowed fields are set? Are types enforced (in other words, don't put an int in a string field?)</i>	Yes, based on the OpenAPI v3.0 validation schema (GA in 1.16).	Yes

Common Features

When you create a custom resource, either via a CRD or an AA, you get many features for your API, compared to implementing it outside the Kubernetes platform:

Feature	What it does
CRUD	<i>The new endpoints support CRUD basic operations via HTTP and kubectl</i>
Watch	<i>The new endpoints support Kubernetes Watch operations via HTTP</i>
Discovery	<i>Clients like kubectl and dashboard automatically offer list, display, and field edit operations on your resources</i>
json-patch	<i>The new endpoints support PATCH with Content-Type: application/json-patch+json</i>
merge-patch	<i>The new endpoints support PATCH with Content-Type: application/merge-patch+json</i>
HTTPS	<i>The new endpoints uses HTTPS</i>
Built-in Authentication	<i>Access to the extension uses the core API server (aggregation layer) for authentication</i>
Built-in Authorization	<i>Access to the extension can reuse the authorization used by the core API server; for example, RBAC.</i>
Finalizers	<i>Block deletion of extension resources until external cleanup happens.</i>
Admission Webhooks	<i>Set default values and validate extension resources during any create/update/delete operation.</i>
UI/CLI Display	<i>Kubectl, dashboard can display extension resources.</i>
Unset versus Empty	<i>Clients can distinguish unset fields from zero-valued fields.</i>
Client Libraries Generation	<i>Kubernetes provides generic client libraries, as well as tools to generate type-specific client libraries.</i>
Labels and annotations	<i>Common metadata across objects that tools know how to edit for core and custom resources.</i>

Preparing to install a custom resource

There are several points to be aware of before adding a custom resource to your cluster.

Third party code and new points of failure

While creating a CRD does not automatically add any new points of failure (for example, by causing third party code to run on your API server), packages (for example, Charts) or other installation bundles often include CRDs as well as a Deployment of third-party code that implements the business logic for a new custom resource.

Installing an Aggregated API server always involves running a new Deployment.

Storage

Custom resources consume storage space in the same way that ConfigMaps do. Creating too many custom resources may overload your API server's storage space.

Aggregated API servers may use the same storage as the main API server, in which case the same warning applies.

Authentication, authorization, and auditing

CRDs always use the same authentication, authorization, and audit logging as the built-in resources of your API server.

If you use RBAC for authorization, most RBAC roles will not grant access to the new resources (except the cluster-admin role or any role created with wildcard rules). You'll need to explicitly grant access to the new resources. CRDs and Aggregated APIs often come bundled with new role definitions for the types they add.

Aggregated API servers may or may not use the same authentication, authorization, and auditing as the primary API server.

Accessing a custom resource

Kubernetes [client libraries](#) can be used to access custom resources. Not all client libraries support custom resources. The Go and Python client libraries do.

When you add a custom resource, you can access it using:

- `kubectl`
- The kubernetes dynamic client.
- A REST client that you write.
- A client generated using [Kubernetes client generation tools](#) (generating one is an advanced undertaking, but some projects may provide a client along with the CRD or AA).

What's next

- Learn how to [Extend the Kubernetes API with the aggregation layer](#).
- Learn how to [Extend the Kubernetes API with CustomResourceDefinition](#).

Feedback

Was this page helpful?

Yes No

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

Last modified October 13, 2020 at 12:41 AM PST: [Move API overview to be a Docsy section overview \(3edb97057\)](#)

[Edit this page](#) [Create child page](#) [Create an issue](#)

- [Custom resources](#)
- [Custom controllers](#)
- [Should I add a custom resource to my Kubernetes Cluster?](#)
 - [Declarative APIs](#)
- [Should I use a configMap or a custom resource?](#)
- [Adding custom resources](#)
- [CustomResourceDefinitions](#)
- [API server aggregation](#)
- [Choosing a method for adding custom resources](#)
 - [Comparing ease of use](#)
 - [Advanced features and flexibility](#)
 - [Common Features](#)
- [Preparing to install a custom resource](#)
 - [Third party code and new points of failure](#)
 - [Storage](#)
 - [Authentication, authorization, and auditing](#)
- [Accessing a custom resource](#)
- [What's next](#)

Extending the Kubernetes API with the aggregation layer

The aggregation layer allows Kubernetes to be extended with additional APIs, beyond what is offered by the core Kubernetes APIs. The additional APIs can either be ready-made solutions such as [service-catalog](#), or APIs that you develop yourself.

The aggregation layer is different from [Custom Resources](#), which are a way to make the [kube-apiserver](#) recognise new kinds of object.

Aggregation layer

The aggregation layer runs in-process with the kube-apiserver. Until an extension resource is registered, the aggregation layer will do nothing. To

register an API, you add an `APIService` object, which "claims" the URL path in the Kubernetes API. At that point, the aggregation layer will proxy anything sent to that API path (e.g. `/apis/myextension.mycompany.io/v1/`) to the registered `APIService`.

The most common way to implement the `APIService` is to run an extension API server in Pod(s) that run in your cluster. If you're using the extension API server to manage resources in your cluster, the extension API server (also written as "extension-apiserver") is typically paired with one or more [controllers](#). The `apiserver-builder` library provides a skeleton for both extension API servers and the associated controller(s).

Response latency

Extension API servers should have low latency networking to and from the `kube-apiserver`. Discovery requests are required to round-trip from the `kube-apiserver` in five seconds or less.

If your extension API server cannot achieve that latency requirement, consider making changes that let you meet it. You can also set the `EnableAggregatedDiscoveryTimeout=false` [feature gate](#) on the `kube-apiserver` to disable the timeout restriction. This deprecated feature gate will be removed in a future release.

What's next

- To get the aggregator working in your environment, [configure the aggregation layer](#).
- Then, [setup an extension api-server](#) to work with the aggregation layer.
- Also, learn how to [extend the Kubernetes API using Custom Resource Definitions](#).
- Read the specification for [APIService](#)

Feedback

Was this page helpful?

Yes No

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

Last modified July 17, 2020 at 4:11 PM PST: [Fix links in concepts section \(2\)](#) ([c8f470487](#))

[Edit this page](#) [Create child page](#) [Create an issue](#)

- [Aggregation layer](#)
 - [Response latency](#)
- [What's next](#)

Compute, Storage, and Networking Extensions

Network Plugins

Device Plugins

Use the Kubernetes device plugin framework to implement plugins for GPUs, NICs, FPGAs, InfiniBand, and similar resources that require vendor-specific setup.

Network Plugins

Network plugins in Kubernetes come in a few flavors:

- CNI plugins: adhere to the [Container Network Interface](#) (CNI) specification, designed for interoperability.
 - Kubernetes follows the [v0.4.0](#) release of the CNI specification.
- Kubenet plugin: implements basic *cbr0* using the bridge and host-local CNI plugins

Installation

The kubelet has a single default network plugin, and a default network common to the entire cluster. It probes for plugins when it starts up, remembers what it finds, and executes the selected plugin at appropriate times in the pod lifecycle (this is only true for Docker, as CRI manages its own CNI plugins). There are two Kubelet command line parameters to keep in mind when using plugins:

- *cni-bin-dir*: Kubelet probes this directory for plugins on startup
- *network-plugin*: The network plugin to use from *cni-bin-dir*. It must match the name reported by a plugin probed from the plugin directory. For CNI plugins, this is simply "cni".

Network Plugin Requirements

Besides providing the [NetworkPlugin interface](#) to configure and clean up pod networking, the plugin may also need specific support for kube-proxy. The iptables proxy obviously depends on iptables, and the plugin may need to ensure that container traffic is made available to iptables. For example, if the plugin connects containers to a Linux bridge, the plugin must set the net /bridge/bridge-nf-call-iptables sysctl to 1 to ensure that the iptables proxy functions correctly. If the plugin does not use a Linux bridge (but

(instead something like Open vSwitch or some other mechanism) it should ensure container traffic is appropriately routed for the proxy.

By default if no kubelet network plugin is specified, the noop plugin is used, which sets net/bridge/bridge-nf-call-iptables=1 to ensure simple configurations (like Docker with a bridge) work correctly with the iptables proxy.

CNI

The CNI plugin is selected by passing Kubelet the --network-plugin=cni command-line option. Kubelet reads a file from --cni-conf-dir (default /etc/cni/net.d) and uses the CNI configuration from that file to set up each pod's network. The CNI configuration file must match the [CNI specification](#), and any required CNI plugins referenced by the configuration must be present in --cni-bin-dir (default /opt/cni/bin).

If there are multiple CNI configuration files in the directory, the kubelet uses the configuration file that comes first by name in lexicographic order.

In addition to the CNI plugin specified by the configuration file, Kubernetes requires the standard CNI [lo](#) plugin, at minimum version 0.2.0

Support hostPort

The CNI networking plugin supports hostPort. You can use the official [portmap](#) plugin offered by the CNI plugin team or use your own plugin with portMapping functionality.

If you want to enable hostPort support, you must specify portMappings capability in your cni-conf-dir. For example:

```
{  
  "name": "k8s-pod-network",  
  "cniVersion": "0.3.0",  
  "plugins": [  
    {  
      "type": "calico",  
      "log_level": "info",  
      "datastore_type": "kubernetes",  
      "nodename": "127.0.0.1",  
      "ipam": {  
        "type": "host-local",  
        "subnet": "usePodCidr"  
      },  
      "policy": {  
        "type": "k8s"  
      },  
      "kubernetes": {  
        "kubeconfig": "/etc/cni/net.d/calico-kubeconfig"  
      }  
    },  
  ],  
}
```

```
{
  "type": "portmap",
  "capabilities": {"portMappings": true}
}
]
```

Support traffic shaping

Experimental Feature

The CNI networking plugin also supports pod ingress and egress traffic shaping. You can use the official [bandwidth](#) plugin offered by the CNI plugin team or use your own plugin with bandwidth control functionality.

If you want to enable traffic shaping support, you must add the `bandwidth` plugin to your CNI configuration file (default `/etc/cni/net.d`) and ensure that the binary is included in your CNI bin dir (default `/opt/cni/bin`).

```
{
  "name": "k8s-pod-network",
  "cniVersion": "0.3.0",
  "plugins": [
    {
      "type": "calico",
      "log_level": "info",
      "datastore_type": "kubernetes",
      "nodename": "127.0.0.1",
      "ipam": {
        "type": "host-local",
        "subnet": "usePodCidr"
      },
      "policy": {
        "type": "k8s"
      },
      "kubernetes": {
        "kubeconfig": "/etc/cni/net.d/calico-kubeconfig"
      }
    },
    {
      "type": "bandwidth",
      "capabilities": {"bandwidth": true}
    }
  ]
}
```

Now you can add the `kubernetes.io/ingress-bandwidth` and `kubernetes.io/egress-bandwidth` annotations to your pod. For example:

```
apiVersion: v1
kind: Pod
metadata:
```

```
annotations:
```

```
  kubernetes.io/ingress-bandwidth: 1M
```

```
  kubernetes.io/egress-bandwidth: 1M
```

kubenet

Kubenet is a very basic, simple network plugin, on Linux only. It does not, of itself, implement more advanced features like cross-node networking or network policy. It is typically used together with a cloud provider that sets up routing rules for communication between nodes, or in single-node environments.

Kubenet creates a Linux bridge named `cbr0` and creates a veth pair for each pod with the host end of each pair connected to `cbr0`. The pod end of the pair is assigned an IP address allocated from a range assigned to the node either through configuration or by the controller-manager. `cbr0` is assigned an MTU matching the smallest MTU of an enabled normal interface on the host.

The plugin requires a few things:

- The standard CNI bridge, `lo` and `host-local` plugins are required, at minimum version 0.2.0. Kubenet will first search for them in `/opt/cni/bin`. Specify `cni-bin-dir` to supply additional search path. The first found match will take effect.
- Kubelet must be run with the `--network-plugin=kubenet` argument to enable the plugin
- Kubelet should also be run with the `--non-masquerade-cidr=<clusterCidr>` argument to ensure traffic to IPs outside this range will use IP masquerade.
- The node must be assigned an IP subnet through either the `--pod-cidr` kubelet command-line option or the `--allocate-node-cidrs=true --cluster-cidr=<cidr>` controller-manager command-line options.

Customizing the MTU (with kubenet)

The MTU should always be configured correctly to get the best networking performance. Network plugins will usually try to infer a sensible MTU, but sometimes the logic will not result in an optimal MTU. For example, if the Docker bridge or another interface has a small MTU, kubenet will currently select that MTU. Or if you are using IPSEC encapsulation, the MTU must be reduced, and this calculation is out-of-scope for most network plugins.

Where needed, you can specify the MTU explicitly with the `network-plugin-mtu` kubelet option. For example, on AWS the `eth0` MTU is typically 9001, so you might specify `--network-plugin-mtu=9001`. If you're using IPSEC you might reduce it to allow for encapsulation overhead; for example: `--network-plugin-mtu=8873`.

*This option is provided to the network-plugin; currently **only kubenet supports network-plugin-mtu**.*

Usage Summary

- `--network-plugin=cni` specifies that we use the `cni` network plugin with actual CNI plugin binaries located in `--cni-bin-dir` (default `/opt/cni/bin`) and CNI plugin configuration located in `--cni-conf-dir` (default `/etc/cni/net.d`).
- `--network-plugin=kubenet` specifies that we use the `kubenet` network plugin with CNI bridge and host-local plugins placed in `/opt/cni/bin` or `cni-bin-dir`.
- `--network-plugin-mtu=9001` specifies the MTU to use, currently only used by the `kubenet` network plugin.

What's next

Feedback

Was this page helpful?

Yes No

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

Last modified October 13, 2020 at 7:22 AM PST: [Make the CNI usage more accurate \(c3a9924bb\)](#)

[Edit this page](#) [Create child page](#) [Create an issue](#)

- [Installation](#)
- [Network Plugin Requirements](#)
 - [CNI](#)
 - [kubenet](#)
 - [Customizing the MTU \(with kubenet\)](#)
- [Usage Summary](#)
- [What's next](#)

Device Plugins

Use the Kubernetes device plugin framework to implement plugins for GPUs, NICs, FPGAs, InfiniBand, and similar resources that require vendor-specific setup.

FEATURE STATE: Kubernetes v1.10 [beta]

Kubernetes provides a [device plugin framework](#) that you can use to advertise system hardware resources to the [Kubelet](#).

Instead of customizing the code for Kubernetes itself, vendors can implement a device plugin that you deploy either manually or as a

[DaemonSet](#). The targeted devices include GPUs, high-performance NICs, FPGAs, InfiniBand adapters, and other similar computing resources that may require vendor specific initialization and setup.

Device plugin registration

The kubelet exports a *Registration* gRPC service:

```
service Registration {
    rpc Register(RegisterRequest) returns (Empty) {}
}
```

A device plugin can register itself with the kubelet through this gRPC service. During the registration, the device plugin needs to send:

- The name of its Unix socket.
- The Device Plugin API version against which it was built.
- The ResourceName it wants to advertise. Here ResourceName needs to follow the [extended resource naming scheme](#) as vendor-domain/resource-type. (For example, an NVIDIA GPU is advertised as nvidia.com/gpu.)

Following a successful registration, the device plugin sends the kubelet the list of devices it manages, and the kubelet is then in charge of advertising those resources to the API server as part of the kubelet node status update. For example, after a device plugin registers hardware-vendor.example/foo with the kubelet and reports two healthy devices on a node, the node status is updated to advertise that the node has 2 "Foo" devices installed and available.

Then, users can request devices in a [Container](#) specification as they request other types of resources, with the following limitations:

- Extended resources are only supported as integer resources and cannot be overcommitted.
- Devices cannot be shared among Containers.

Suppose a Kubernetes cluster is running a device plugin that advertises resource hardware-vendor.example/foo on certain nodes. Here is an example of a pod requesting this resource to run a demo workload:

```
---
apiVersion: v1
kind: Pod
metadata:
  name: demo-pod
spec:
  containers:
    - name: demo-container-1
      image: k8s.gcr.io/pause:2.0
      resources:
        limits:
```

```

hardware-vendor.example/foo: 2
#
# This Pod needs 2 of the hardware-vendor.example/foo devices
# and can only schedule onto a Node that's able to satisfy
# that need.
#
# If the Node has more than 2 of those devices available, the
# remainder would be available for other Pods to use.

```

Device plugin implementation

The general workflow of a device plugin includes the following steps:

- *Initialization.* During this phase, the device plugin performs vendor specific initialization and setup to make sure the devices are in a ready state.
- The plugin starts a gRPC service, with a Unix socket under host path /var/lib/kubelet/device-plugins/, that implements the following interfaces:

```

service DevicePlugin {
    // GetDevicePluginOptions returns options to be
    // communicated with Device Manager.
    rpc GetDevicePluginOptions(Empty) returns
    (DevicePluginOptions) {}

    // ListAndWatch returns a stream of List of Devices
    // Whenever a Device state change or a Device
    // disappears, ListAndWatch
    // returns the new list
    rpc ListAndWatch(Empty) returns (stream
    ListAndWatchResponse) {}

    // Allocate is called during container creation so
    // that the Device
    // Plugin can run device specific operations and
    // instruct Kubelet
    // of the steps to make the Device available in the
    // container
    rpc Allocate(AllocateRequest) returns
    (AllocateResponse) {}

    // GetPreferredAllocation returns a preferred set of
    // devices to allocate
    // from a list of available ones. The resulting
    // preferred allocation is not
    // guaranteed to be the allocation ultimately
    // performed by the
    // devicemanager. It is only designed to help the
    // devicemanager make a more

```

```

    // informed allocation decision when possible.
    rpc GetPreferredAllocation(PreferredAllocationRequest)
returns (PreferredAllocationResponse) {}

    // PreStartContainer is called, if indicated by Device
    // Plugin during registration phase,
    // before each container start. Device plugin can run
    // device specific operations
    // such as resetting the device before making devices
    // available to the container.
    rpc PreStartContainer(PreStartContainerRequest)
returns (PreStartContainerResponse) {}
}

```

Note: Plugins are not required to provide useful implementations for `GetPreferredAllocation()` or `PreStartContainer()`. Flags indicating which (if any) of these calls are available should be set in the `DevicePluginOptions` message sent back by a call to `GetDevicePluginOptions()`. The `kubelet` will always call `GetDevicePluginOptions()` to see which optional functions are available, before calling any of them directly.

- The plugin registers itself with the `kubelet` through the Unix socket at host path `/var/lib/kubelet/device-plugins/kubelet.sock`.
- After successfully registering itself, the device plugin runs in serving mode, during which it keeps monitoring device health and reports back to the `kubelet` upon any device state changes. It is also responsible for serving `Allocate` gRPC requests. During `Allocate`, the device plugin may do device-specific preparation; for example, GPU cleanup or QRNG initialization. If the operations succeed, the device plugin returns an `AllocateResponse` that contains container runtime configurations for accessing the allocated devices. The `kubelet` passes this information to the container runtime.

Handling kubelet restarts

A device plugin is expected to detect `kubelet` restarts and re-register itself with the new `kubelet` instance. In the current implementation, a new `kubelet` instance deletes all the existing Unix sockets under `/var/lib/kubelet/device-plugins` when it starts. A device plugin can monitor the deletion of its Unix socket and re-register itself upon such an event.

Device plugin deployment

You can deploy a device plugin as a `DaemonSet`, as a package for your node's operating system, or manually.

The canonical directory `/var/lib/kubelet/device-plugins` requires privileged access, so a device plugin must run in a privileged security

context. If you're deploying a device plugin as a DaemonSet, /var/lib/kubelet/device-plugins must be mounted as a [Volume](#) in the plugin's PodSpec.

If you choose the DaemonSet approach you can rely on Kubernetes to: place the device plugin's Pod onto Nodes, to restart the daemon Pod after failure, and to help automate upgrades.

API compatibility

Kubernetes device plugin support is in beta. The API may change before stabilization, in incompatible ways. As a project, Kubernetes recommends that device plugin developers:

- *Watch for changes in future releases.*
- *Support multiple versions of the device plugin API for backward/forward compatibility.*

If you enable the DevicePlugins feature and run device plugins on nodes that need to be upgraded to a Kubernetes release with a newer device plugin API version, upgrade your device plugins to support both versions before upgrading these nodes. Taking that approach will ensure the continuous functioning of the device allocations during the upgrade.

Monitoring Device Plugin Resources

FEATURE STATE: Kubernetes v1.15 [beta]

In order to monitor resources provided by device plugins, monitoring agents need to be able to discover the set of devices that are in-use on the node and obtain metadata to describe which container the metric should be associated with. [Prometheus](#) metrics exposed by device monitoring agents should follow the [Kubernetes Instrumentation Guidelines](#), identifying containers using pod, namespace, and container prometheus labels.

The kubelet provides a gRPC service to enable discovery of in-use devices, and to provide metadata for these devices:

```
// PodResourcesLister is a service provided by the kubelet that
// provides information about the
// node resources consumed by pods and containers on the node
service PodResourcesLister {
    rpc List(ListPodResourcesRequest) returns
(ListPodResourcesResponse) {}
}
```

The gRPC service is served over a unix socket at /var/lib/kubelet/pod-resources/kubelet.sock. Monitoring agents for device plugin resources can be deployed as a daemon, or as a DaemonSet. The canonical directory /var/lib/kubelet/pod-resources requires privileged access, so monitoring agents must run in a privileged security context. If a device monitoring

agent is running as a DaemonSet, /var/lib/kubelet/pod-resources must be mounted as a [Volume](#) in the plugin's [PodSpec](#).

Support for the "PodResources service" requires [KubeletPodResources feature gate](#) to be enabled. It is enabled by default starting with Kubernetes 1.15 and is v1 since Kubernetes 1.20.

Device Plugin integration with the Topology Manager

FEATURE STATE: Kubernetes v1.18 [beta]

The Topology Manager is a Kubelet component that allows resources to be co-ordinated in a Topology aligned manner. In order to do this, the Device Plugin API was extended to include a `TopologyInfo` struct.

```
message TopologyInfo {  
    repeated NUMANode nodes = 1;  
}  
  
message NUMANode {  
    int64 ID = 1;  
}
```

Device Plugins that wish to leverage the Topology Manager can send back a populated `TopologyInfo` struct as part of the device registration, along with the device IDs and the health of the device. The device manager will then use this information to consult with the Topology Manager and make resource assignment decisions.

`TopologyInfo` supports a `nodes` field that is either `nil` (the default) or a list of NUMA nodes. This lets the Device Plugin publish that can span NUMA nodes.

An example `TopologyInfo` struct populated for a device by a Device Plugin:

```
pluginapi.Device{ID: "25102017", Health: pluginapi.Healthy,  
Topology:&pluginapi.TopologyInfo{Nodes:  
[]*pluginapi.NUMANode{&pluginapi.NUMANode{ID: 0},}}}}
```

Device plugin examples

Here are some examples of device plugin implementations:

- The [AMD GPU device plugin](#)
- The [Intel device plugins](#) for Intel GPU, FPGA and QuickAssist devices
- The [KubeVirt device plugins](#) for hardware-assisted virtualization
- The [NVIDIA GPU device plugin](#)
 - Requires [nvidia-docker](#) 2.0, which allows you to run GPU-enabled Docker containers.
- The [NVIDIA GPU device plugin for Container-Optimized OS](#)

- The [RDMA device plugin](#)
- The [Solarflare device plugin](#)
- The [SR-IOV Network device plugin](#)
- The [Xilinx FPGA device plugins](#) for Xilinx FPGA devices

What's next

- Learn about [scheduling GPU resources using device plugins](#)
- Learn about [advertising extended resources](#) on a node
- Read about using [hardware acceleration for TLS ingress](#) with Kubernetes
- Learn about the [Topology Manager](#)

Feedback

Was this page helpful?

Yes No

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

Last modified October 29, 2020 at 6:43 PM PST: [Graduate KubeletPodResources to GA \(f2ef3d0e8\)](#)

[Edit this page](#) [Create child page](#) [Create an issue](#)

- [Device plugin registration](#)
- [Device plugin implementation](#)
 - [Handling kubelet restarts](#)
- [Device plugin deployment](#)
- [API compatibility](#)
- [Monitoring Device Plugin Resources](#)
- [Device Plugin integration with the Topology Manager](#)
- [Device plugin examples](#)
- [What's next](#)

Operator pattern

Operators are software extensions to Kubernetes that make use of [custom resources](#) to manage applications and their components. Operators follow Kubernetes principles, notably the [control loop](#).

Motivation

The Operator pattern aims to capture the key aim of a human operator who is managing a service or set of services. Human operators who look after

specific applications and services have deep knowledge of how the system ought to behave, how to deploy it, and how to react if there are problems.

People who run workloads on Kubernetes often like to use automation to take care of repeatable tasks. The Operator pattern captures how you can write code to automate a task beyond what Kubernetes itself provides.

Operators in Kubernetes

Kubernetes is designed for automation. Out of the box, you get lots of built-in automation from the core of Kubernetes. You can use Kubernetes to automate deploying and running workloads, and you can automate how Kubernetes does that.

Kubernetes' [controllers](#) concept lets you extend the cluster's behaviour without modifying the code of Kubernetes itself. Operators are clients of the Kubernetes API that act as controllers for a [Custom Resource](#).

An example Operator

Some of the things that you can use an operator to automate include:

- *deploying an application on demand*
- *taking and restoring backups of that application's state*
- *handling upgrades of the application code alongside related changes such as database schemas or extra configuration settings*
- *publishing a Service to applications that don't support Kubernetes APIs to discover them*
- *simulating failure in all or part of your cluster to test its resilience*
- *choosing a leader for a distributed application without an internal member election process*

What might an Operator look like in more detail? Here's an example in more detail:

1. *A custom resource named SampleDB, that you can configure into the cluster.*
2. *A Deployment that makes sure a Pod is running that contains the controller part of the operator.*
3. *A container image of the operator code.*
4. *Controller code that queries the control plane to find out what SampleDB resources are configured.*
5. *The core of the Operator is code to tell the API server how to make reality match the configured resources.*
 - *If you add a new SampleDB, the operator sets up PersistentVolumeClaims to provide durable database storage, a StatefulSet to run SampleDB and a Job to handle initial configuration.*
 - *If you delete it, the Operator takes a snapshot, then makes sure that the StatefulSet and Volumes are also removed.*

6. The operator also manages regular database backups. For each `SampleDB` resource, the operator determines when to create a Pod that can connect to the database and take backups. These Pods would rely on a `ConfigMap` and / or a `Secret` that has database connection details and credentials.
7. Because the Operator aims to provide robust automation for the resource it manages, there would be additional supporting code. For this example, code checks to see if the database is running an old version and, if so, creates Job objects that upgrade it for you.

Deploying Operators

The most common way to deploy an Operator is to add the Custom Resource Definition and its associated Controller to your cluster. The Controller will normally run outside of the [control plane](#), much as you would run any containerized application. For example, you can run the controller in your cluster as a Deployment.

Using an Operator

Once you have an Operator deployed, you'd use it by adding, modifying or deleting the kind of resource that the Operator uses. Following the above example, you would set up a Deployment for the Operator itself, and then:

```
kubectl get SampleDB          # find configured
databases

kubectl edit SampleDB/example-database # manually change some
settings
```

and that's it! The Operator will take care of applying the changes as well as keeping the existing service in good shape.

Writing your own Operator

If there isn't an Operator in the ecosystem that implements the behavior you want, you can code your own. In [What's next](#) you'll find a few links to libraries and tools you can use to write your own cloud native Operator.

You also implement an Operator (that is, a Controller) using any language / runtime that can act as a [client for the Kubernetes API](#).

What's next

- Learn more about [Custom Resources](#)
- Find ready-made operators on [OperatorHub.io](#) to suit your use case
- Use existing tools to write your own operator, eg:
 - using [KUDO](#) (Kubernetes Universal Declarative Operator)
 - using [kubebuilder](#)

- using [Metacontroller](#) along with WebHooks that you implement yourself
- using the [Operator Framework](#)
- [Publish](#) your operator for other people to use
- Read [CoreOS' original article](#) that introduced the Operator pattern
- Read an [article](#) from Google Cloud about best practices for building Operators

Feedback

Was this page helpful?

Yes No

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

Last modified October 22, 2020 at 2:24 PM PST: [Fix links in concepts section \(070023b24\)](#)

[Edit this page](#) [Create child page](#) [Create an issue](#)

- [Motivation](#)
- [Operators in Kubernetes](#)
- [An example Operator](#)
- [Deploying Operators](#)
- [Using an Operator](#)
- [Writing your own Operator](#)
- [What's next](#)

Service Catalog

Service Catalog is an extension API that enables applications running in Kubernetes clusters to easily use external managed software offerings, such as a datastore service offered by a cloud provider.

It provides a way to list, provision, and bind with external [Managed Services](#) from [Service Brokers](#) without needing detailed knowledge about how those services are created or managed.

A service broker, as defined by the [Open service broker API spec](#), is an endpoint for a set of managed services offered and maintained by a third-party, which could be a cloud provider such as AWS, GCP, or Azure. Some examples of managed services are Microsoft Azure Cloud Queue, Amazon Simple Queue Service, and Google Cloud Pub/Sub, but they can be any software offering that can be used by an application.

Using Service Catalog, a [cluster operator](#) can browse the list of managed services offered by a service broker, provision an instance of a managed

service, and bind with it to make it available to an application in the Kubernetes cluster.

Example use case

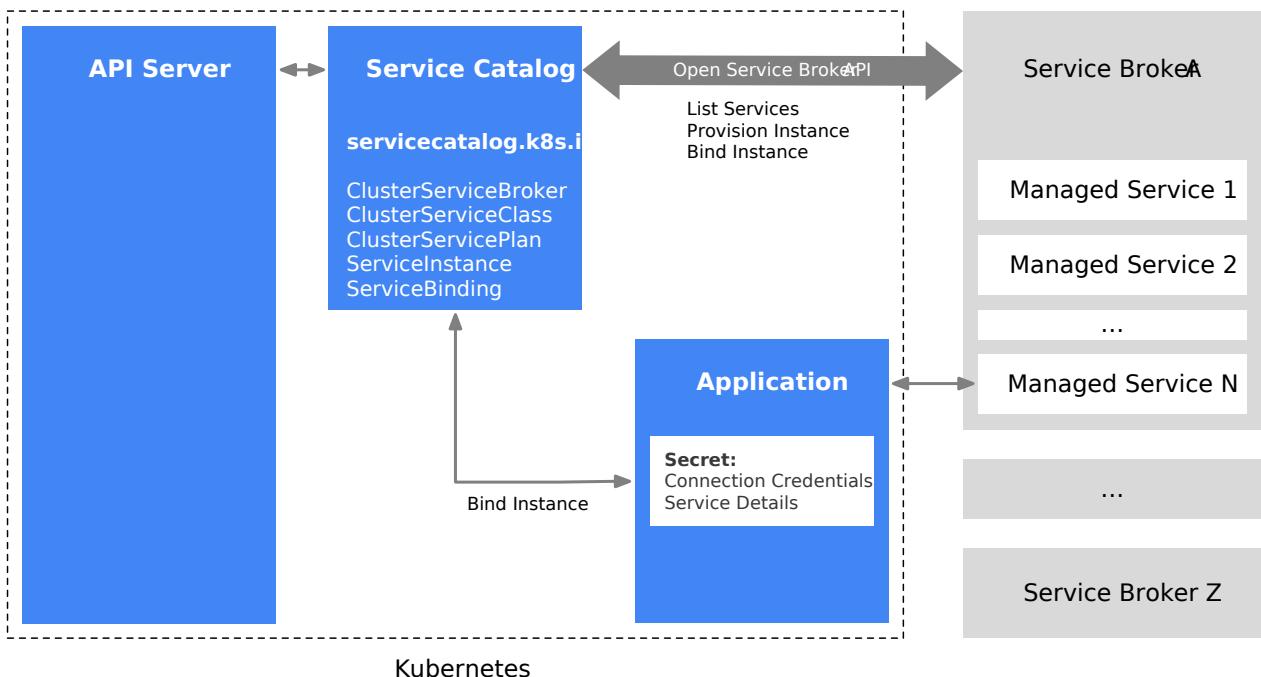
An [application developer](#) wants to use message queuing as part of their application running in a Kubernetes cluster. However, they do not want to deal with the overhead of setting such a service up and administering it themselves. Fortunately, there is a cloud provider that offers message queuing as a managed service through its service broker.

A cluster operator can setup Service Catalog and use it to communicate with the cloud provider's service broker to provision an instance of the message queuing service and make it available to the application within the Kubernetes cluster. The application developer therefore does not need to be concerned with the implementation details or management of the message queue. The application can simply use it as a service.

Architecture

Service Catalog uses the [Open service broker API](#) to communicate with service brokers, acting as an intermediary for the Kubernetes API Server to negotiate the initial provisioning and retrieve the credentials necessary for the application to use a managed service.

It is implemented as an extension API server and a controller, using etcd for storage. It also uses the [aggregation layer](#) available in Kubernetes 1.7+ to present its API.



API Resources

Service Catalog installs the `servicecatalog.k8s.io` API and provides the following Kubernetes resources:

- *`ClusterServiceBroker`: An in-cluster representation of a service broker, encapsulating its server connection details. These are created and managed by cluster operators who wish to use that broker server to make new types of managed services available within their cluster.*
- *`ClusterServiceClass`: A managed service offered by a particular service broker. When a new `ClusterServiceBroker` resource is added to the cluster, the Service Catalog controller connects to the service broker to obtain a list of available managed services. It then creates a new `ClusterServiceClass` resource corresponding to each managed service.*
- *`ClusterServicePlan`: A specific offering of a managed service. For example, a managed service may have different plans available, such as a free tier or paid tier, or it may have different configuration options, such as using SSD storage or having more resources. Similar to `ClusterServiceClass`, when a new `ClusterServiceBroker` is added to the cluster, Service Catalog creates a new `ClusterServicePlan` resource corresponding to each Service Plan available for each managed service.*
- *`ServiceInstance`: A provisioned instance of a `ClusterServiceClass`. These are created by cluster operators to make a specific instance of a managed service available for use by one or more in-cluster applications. When a new `ServiceInstance` resource is created, the Service Catalog controller connects to the appropriate service broker and instruct it to provision the service instance.*
- *`ServiceBinding`: Access credentials to a `ServiceInstance`. These are created by cluster operators who want their applications to make use of a `ServiceInstance`. Upon creation, the Service Catalog controller creates a Kubernetes Secret containing connection details and credentials for the Service Instance, which can be mounted into Pods.*

Authentication

Service Catalog supports these methods of authentication:

- Basic (username/password)
- [OAuth 2.0 Bearer Token](#)

Usage

A cluster operator can use Service Catalog API Resources to provision managed services and make them available within a Kubernetes cluster. The steps involved are:

1. Listing the managed services and Service Plans available from a service broker.
2. Provisioning a new instance of the managed service.
3. Binding to the managed service, which returns the connection credentials.
4. Mapping the connection credentials into the application.

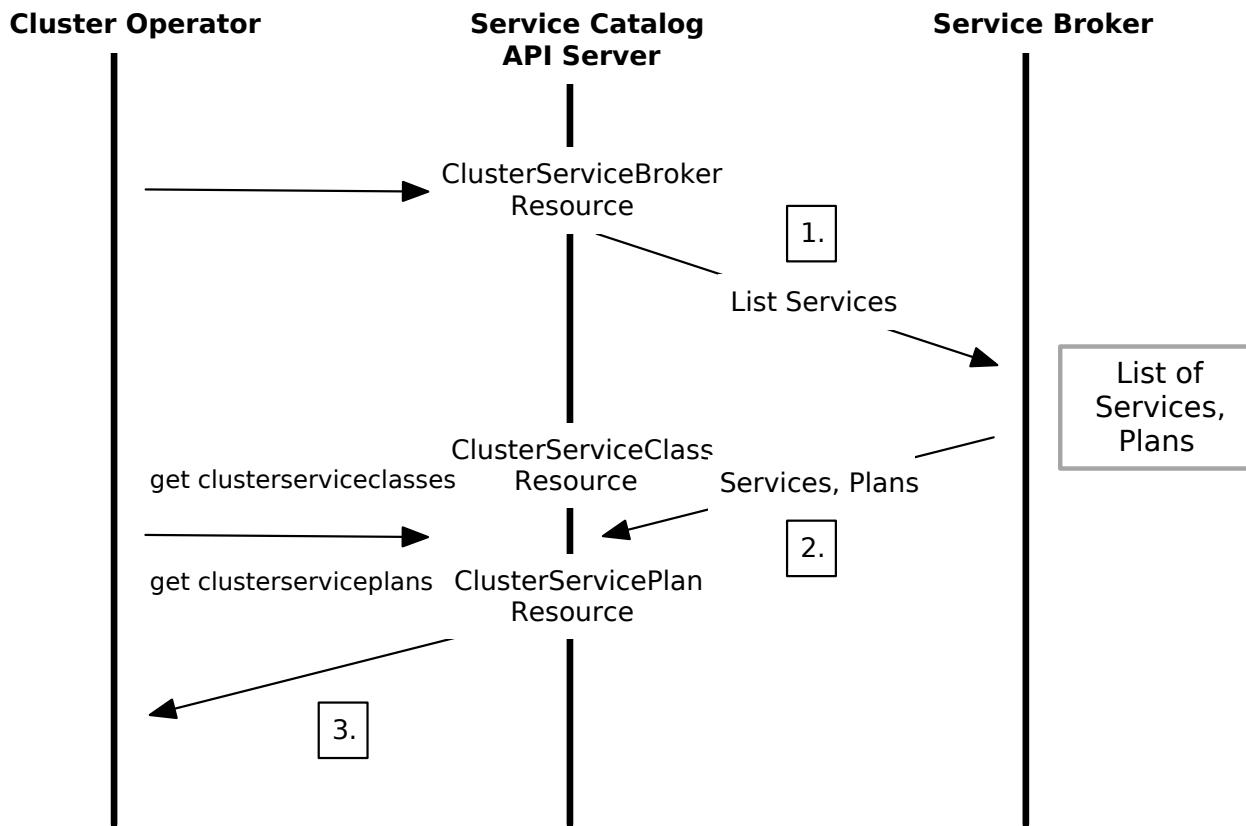
Listing managed services and Service Plans

First, a cluster operator must create a *ClusterServiceBroker* resource within the `servicecatalog.k8s.io` group. This resource contains the URL and connection details necessary to access a service broker endpoint.

This is an example of a *ClusterServiceBroker* resource:

```
apiVersion: servicecatalog.k8s.io/v1beta1
kind: ClusterServiceBroker
metadata:
  name: cloud-broker
spec:
  # Points to the endpoint of a service broker. (This example is
  # not a working URL.)
  url: https://servicebroker.somecloudprovider.com/v1alpha1/
  projects/service-catalog/brokers/default
  #####
  # Additional values can be added here, which may be used to
  # communicate
  # with the service broker, such as bearer token info or a
  caBundle for TLS.
  #####
```

The following is a sequence diagram illustrating the steps involved in listing managed services and Plans available from a service broker:



1. Once the *ClusterServiceBroker* resource is added to *Service Catalog*, it triggers a call to the external service broker for a list of available services.
2. The service broker returns a list of available managed services and a list of *Service Plans*, which are cached locally as *ClusterServiceClass* and *ClusterServicePlan* resources respectively.
3. A cluster operator can then get the list of available managed services using the following command:

```
kubectl get clusterserviceclasses -o=custom-columns=SERVICE\ NAME:.metadata.name,EXTERNAL\ NAME:.spec.externalName
```

It should output a list of service names with a format similar to:

SERVICE NAME 4f6e6cf6-ffdd-425f-a2c7-3c9258ad2468 service ...	EXTERNAL NAME cloud-provider-
--	----------------------------------

They can also view the Service Plans available using the following command:

```
kubectl get clusterserviceplans -o=custom-columns=PLAN\ NAME:.metadata.name,EXTERNAL\ NAME:.spec.externalName
```

It should output a list of plan names with a format similar to:

PLAN NAME	EXTERNAL NAME
86064792-7ea2-467b-af93-ac9694d96d52	service-plan-name
...	...

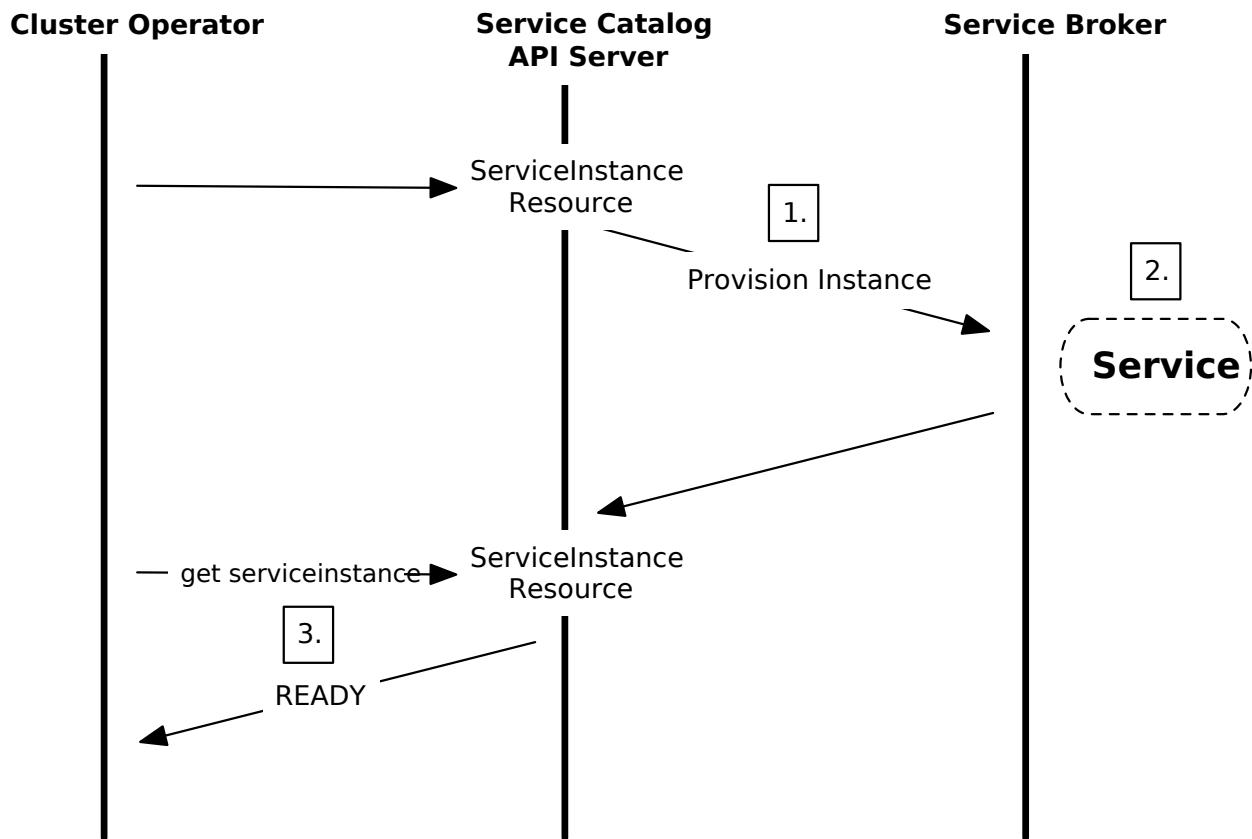
Provisioning a new instance

A cluster operator can initiate the provisioning of a new instance by creating a `ServiceInstance` resource.

This is an example of a `ServiceInstance` resource:

```
apiVersion: servicecatalog.k8s.io/v1beta1
kind: ServiceInstance
metadata:
  name: cloud-queue-instance
  namespace: cloud-apps
spec:
  # References one of the previously returned services
  clusterServiceClassExternalName: cloud-provider-service
  clusterServicePlanExternalName: service-plan-name
  #####
  # Additional parameters can be added here,
  # which may be used by the service broker.
  #####
```

The following sequence diagram illustrates the steps involved in provisioning a new instance of a managed service:



1. When the *ServiceInstance* resource is created, *Service Catalog* initiates a call to the external service broker to provision an instance of the service.
2. The service broker creates a new instance of the managed service and returns an HTTP response.
3. A cluster operator can then check the status of the instance to see if it is ready.

Binding to a managed service

After a new instance has been provisioned, a cluster operator must bind to the managed service to get the connection credentials and service account details necessary for the application to use the service. This is done by creating a *ServiceBinding* resource.

The following is an example of a *ServiceBinding* resource:

```

apiVersion: servicecatalog.k8s.io/v1beta1
kind: ServiceBinding
metadata:
  name: cloud-queue-binding
  namespace: cloud-apps
spec:
  instanceRef:

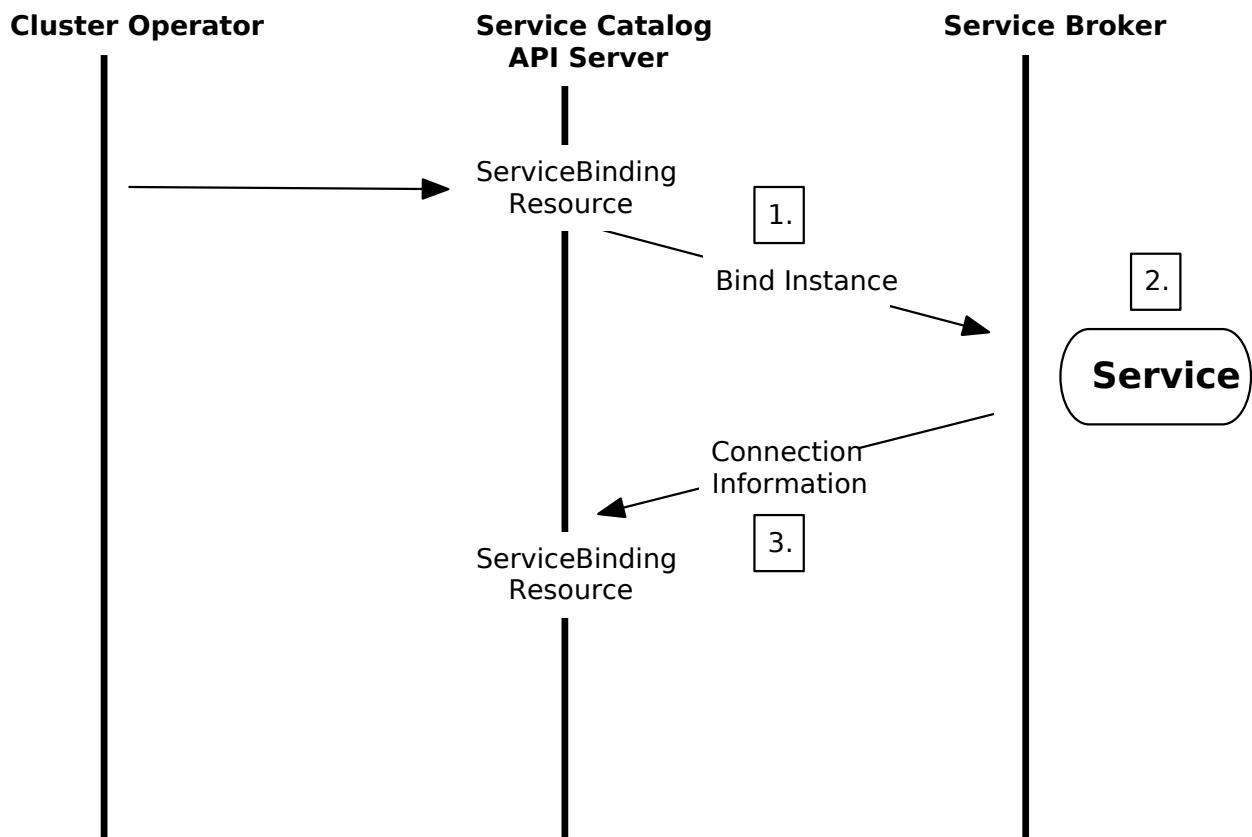
```

```

name: cloud-queue-instance
#####
# Additional information can be added here, such as a
secretName or
# service account parameters, which may be used by the service
broker.
#####

```

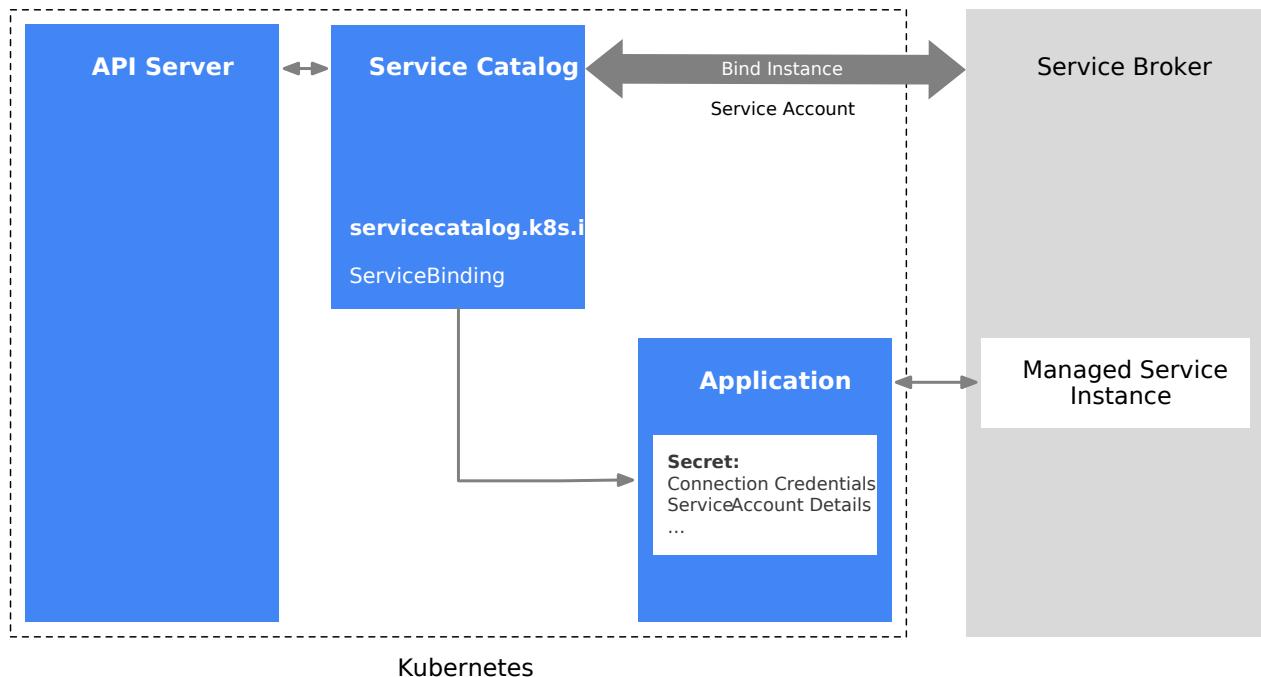
The following sequence diagram illustrates the steps involved in binding to a managed service instance:



1. After the `ServiceBinding` is created, `Service Catalog` makes a call to the external service broker requesting the information necessary to bind with the service instance.
2. The service broker enables the application permissions/roles for the appropriate service account.
3. The service broker returns the information necessary to connect and access the managed service instance. This is provider and service-specific so the information returned may differ between Service Providers and their managed services.

Mapping the connection credentials

After binding, the final step involves mapping the connection credentials and service-specific information into the application. These pieces of information are stored in secrets that the application in the cluster can access and use to connect directly with the managed service.



Pod configuration File

One method to perform this mapping is to use a declarative Pod configuration.

The following example describes how to map service account credentials into the application. A key called `sa-key` is stored in a volume named `provider-cloud-key`, and the application mounts this volume at `/var/secrets/provider/key.json`. The environment variable `PROVIDER_APPLICATION_CREDENTIALS` is mapped from the value of the mounted file.

```
...
spec:
  volumes:
    - name: provider-cloud-key
      secret:
        secretName: sa-key
  containers:
    ...
      volumeMounts:
        - name: provider-cloud-key
          mountPath: /var/secrets/provider
```

```
env:  
  - name: PROVIDER_APPLICATION_CREDENTIALS  
    value: "/var/secrets/provider/key.json"
```

The following example describes how to map secret values into application environment variables. In this example, the messaging queue topic name is mapped from a secret named provider-queue-credentials with a key named `topic` to the environment variable `TOPIC`.

```
...  
env:  
  - name: "TOPIC"  
    valueFrom:  
      secretKeyRef:  
        name: provider-queue-credentials  
        key: topic
```

What's next

- If you are familiar with [Helm Charts](#), [install Service Catalog using Helm](#) into your Kubernetes cluster. Alternatively, you can [install Service Catalog using the SC tool](#).
- View [sample service brokers](#).
- Explore the [kubernetes-sigs/service-catalog](#) project.
- View [svc-cat.io](#).

Feedback

Was this page helpful?

Yes No

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

Last modified October 17, 2020 at 3:21 PM PST: [update kubernetes-incubator references \(a8b6551c2\)](#)

[Edit this page](#) [Create child page](#) [Create an issue](#)

- [Example use case](#)
- [Architecture
 - \[API Resources\]\(#\)
 - \[Authentication\]\(#\)](#)
- [Usage
 - \[Listing managed services and Service Plans\]\(#\)
 - \[Provisioning a new instance\]\(#\)](#)

- [Binding to a managed service](#)
- [Mapping the connection credentials](#)
- [What's next](#)