Edit This Page

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
- Kubernetes Objects
- Kubernetes Control Plane
- What's next

Overview

To work with Kubernetes, you use *Kubernetes API objects* to describe your cluster's *desired state*: what applications or other workloads you want to run, what container images they use, the number of replicas, what network and disk resources you want to make available, and more. You set your desired state by creating objects using the Kubernetes API, typically via the command-line interface, `kubectl`. You can also use the Kubernetes API directly to interact with the cluster and set or modify your desired state.

Once you've set your desired state, the *Kubernetes Control Plane* works to make the cluster's current state match the desired state. To do so, Kubernetes performs a variety of tasks automatically–such as starting or restarting containers, scaling the number of replicas of a given application, and more. The Kubernetes Control Plane consists of a collection of processes running on your cluster:

- The **Kubernetes Master** is a collection of three processes that run on a single node in your cluster, which is designated as the master node. Those processes are: kube-apiserver, kube-controller-manager and kube-scheduler.
- Each individual non-master node in your cluster runs two processes:
  - **kubelet**, which communicates with the Kubernetes Master.
  - **kube-proxy**, a network proxy which reflects Kubernetes networking services on each node.

Kubernetes Objects

Kubernetes contains a number of abstractions that represent the state of your system: deployed containerized applications and workloads, their associated network and disk resources, and other information about what your cluster is

doing. These abstractions are represented by objects in the Kubernetes API; see the Kubernetes Objects overview for more details.

The basic Kubernetes objects include:

- Pod
- Service
- Volume
- Namespace

In addition, Kubernetes contains a number of higher-level abstractions called Controllers. Controllers build upon the basic objects, and provide additional functionality and convenience features. They include:

- ReplicaSet
- Deployment
- StatefulSet
- DaemonSet
- Job

Kubernetes Control Plane

The various parts of the Kubernetes Control Plane, such as the Kubernetes Master and kubelet processes, govern how Kubernetes communicates with your cluster. The Control Plane maintains a record of all of the Kubernetes Objects in the system, and runs continuous control loops to manage those objects' state. At any given time, the Control Plane's control loops will respond to changes in the cluster and work to make the actual state of all the objects in the system match the desired state that you provided.

For example, when you use the Kubernetes API to create a Deployment object, you provide a new desired state for the system. The Kubernetes Control Plane records that object creation, and carries out your instructions by starting the required applications and scheduling them to cluster nodes–thus making the cluster's actual state match the desired state.

Kubernetes Master

The Kubernetes master is responsible for maintaining the desired state for your cluster. When you interact with Kubernetes, such as by using the `kubectl` command-line interface, you're communicating with your cluster's Kubernetes master.

> The "master" refers to a collection of processes managing the cluster state. Typically these processes are all run on a single node in the cluster, and this node is also referred to as the master. The master can also be replicated for availability and redundancy.

Kubernetes Nodes

The nodes in a cluster are the machines (VMs, physical servers, etc) that run your applications and cloud workflows. The Kubernetes master controls each node; you'll rarely interact with nodes directly.

Object Metadata

- Annotations

What's next

If you would like to write a concept page, see Using Page Templates for information about the concept page type and the concept template.

Edit This Page

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
- What's next

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

Every Kubernetes object includes two nested object fields that govern the object's configuration: the object *spec* and the object *status*. The *spec*, which you must provide, describes your *desired state* for the object–the characteristics that you want the object to have. The *status* describes the *actual state* of the object, and is supplied and updated by the Kubernetes system. At any given time, the Kubernetes Control Plane actively manages an object's actual state to match the desired state you supplied.

For example, a Kubernetes 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:

application/deployment.yaml

```
apiVersion: apps/v1 # for versions before 1.9.0 use apps/v1beta2
kind: Deployment
metadata:
  name: nginx-deployment
spec:
  selector:
    matchLabels:
      app: nginx
  replicas: 2 # tells deployment to run 2 pods matching the template
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
      - name: nginx
        image: nginx:1.7.9
        ports:
        - containerPort: 80
```

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

```
$ kubectl create -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`

You'll also need to provide the object `spec` field. 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` object can be found here, and the `spec` format for a `Deployment` object can be found here.

What's next

- Learn about the most important basic Kubernetes objects, such as Pod.

Edit This Page

What is Kubernetes?

This page is an overview of Kubernetes.

- Why do I need Kubernetes and what can it do?
- How is Kubernetes a platform?
- What Kubernetes is not
- Why containers?
- What does Kubernetes mean? K8s?
- What's next

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.

Google open-sourced the Kubernetes project in 2014. Kubernetes builds upon a decade and a half of experience that Google has with running production workloads at scale, combined with best-of-breed ideas and practices from the community.

Why do I need Kubernetes and what can it do?

Kubernetes has a number of features. It can be thought of as:

- a container platform
- a microservices platform
- a portable cloud platform and a lot more.

Kubernetes provides a **container-centric** management environment. It orchestrates computing, networking, and storage infrastructure on behalf of user workloads. This provides much of the simplicity of Platform as a Service (PaaS) with the flexibility of Infrastructure as a Service (IaaS), and enables portability across infrastructure providers.

How is Kubernetes a platform?

Even though Kubernetes provides a lot of functionality, there are always new scenarios that would benefit from new features. Application-specific workflows can be streamlined to accelerate developer velocity. Ad hoc orchestration that is acceptable initially often requires robust automation at scale. This is why Kubernetes was also designed to serve as a platform for building an ecosystem of components and tools to make it easier to deploy, scale, and manage applications.

Labels empower users to organize their resources however they please. Annotations enable users to decorate resources with custom information to facilitate their workflows and provide an easy way for management tools to checkpoint state.

Additionally, the Kubernetes control plane is built upon the same APIs that are available to developers and users. Users can write their own controllers, such as schedulers, with their own APIs that can be targeted by a general-purpose command-line tool.

This design has enabled a number of other systems to build atop Kubernetes.

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, logging, and monitoring. 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:

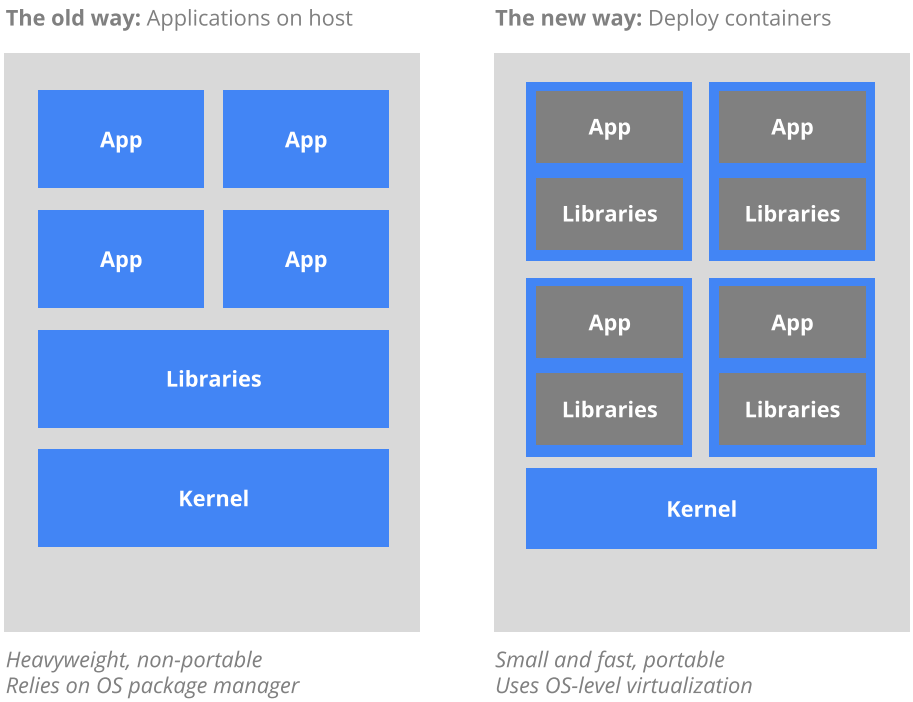
- Does not limit the types of applications supported. Kubernetes aims to support an extremely diverse variety of workloads, including stateless, stateful, and data-processing workloads. If an application can run in a container, it should run great on Kubernetes.
- Does not deploy source code and does 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 (e.g., message buses), data-processing frameworks (for example, Spark), databases (e.g., mysql), caches, nor cluster storage systems (e.g., 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 (e.g., 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 is comprised of 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.

Why containers?

Looking for reasons why you should be using containers?



Heavyweight, non-portable
Relies on OS package manager

Small and fast, portable
Uses OS-level virtualization

The *Old Way* to deploy applications was to install the applications on a host using the operating-system package manager. This had the disadvantage of entangling the applications' executables, configuration, libraries, and lifecycles with each other and with the host OS. One could build immutable virtual-machine images in order to achieve predictable rollouts and rollbacks, but VMs are heavyweight and non-portable.

The *New Way* is to deploy containers based on operating-system-level virtualization rather than hardware virtualization. These containers are isolated from each other and from the host: they have their own filesystems, they can't see each others' processes, and their computational resource usage can be bounded. They are easier to build than VMs, and because they are decoupled from the underlying infrastructure and from the host filesystem, they are portable across clouds and OS distributions.

Because containers are small and fast, one application can be packed in each container image. This one-to-one application-to-image relationship unlocks the full benefits of containers. With containers, immutable container images can be created at build/release time rather than deployment time, since each application doesn't need to be composed with the rest of the application stack, nor married to the production infrastructure environment. Generating container images at build/release time enables a consistent environment to be carried from development into production. Similarly, containers are vastly more transparent than VMs, which facilitates monitoring and management. This is especially

true when the containers' process lifecycles are managed by the infrastructure rather than hidden by a process supervisor inside the container. Finally, with a single application per container, managing the containers becomes tantamount to managing deployment of the application.

Summary of container benefits:

- **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-prem, Google Kubernetes Engine, 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 fat monolithic stack running on one big single-purpose machine.
- **Resource isolation**: Predictable application performance.
- **Resource utilization**: High efficiency and density.

What does Kubernetes mean? K8s?

The name **Kubernetes** originates from Greek, meaning *helmsman* or *pilot*, and is the root of *governor* and cybernetic. *K8s* is an abbreviation derived by replacing the 8 letters "ubernete" with "8".

What's next

- Ready to Get Started?
- For more details, see the Kubernetes Documentation.

Edit This Page

Kubernetes Components

This document outlines the various binary components needed to deliver a functioning Kubernetes cluster.

- Master Components
- Node Components
- Addons

Master Components

Master components provide the cluster's control plane. Master components make global decisions about the cluster (for example, scheduling), and detecting and responding to cluster events (starting up a new pod when a replication controller's 'replicas' field is unsatisfied).

Master components can be run on any machine in the cluster. However, for simplicity, set up scripts typically start all master 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

Component on the master that exposes the Kubernetes API. It is the front-end for the Kubernetes control plane.

It is designed to scale horizontally – that is, it scales by deploying more instances. See Building High-Availability Clusters.

etcd

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

Always have a backup plan for etcd's data for your Kubernetes cluster. For in-depth information on etcd, see etcd documentation.

kube-scheduler

Component on the master that watches newly created pods that have no node assigned, 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

Component on the master that runs controllersA control loop that watches the shared state of the cluster through the apiserver and makes changes attempting to move the current state towards the desired state..

Logically, each controllerA control loop that watches the shared state of the cluster through the apiserver and makes changes attempting to move the current state towards the desired state. 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

cloud-controller-manager runs controllers that interact with the underlying cloud providers. The cloud-controller-manager binary is an alpha feature introduced in Kubernetes release 1.6.

cloud-controller-manager runs cloud-provider-specific controller loops only. You must disable these controller loops in the kube-controller-manager. You can disable the controller loops by setting the `--cloud-provider` flag to `external` when starting the kube-controller-manager.

cloud-controller-manager allows cloud vendors code and the Kubernetes core to evolve independent of each other. In prior releases, the core Kubernetes code was dependent upon cloud-provider-specific code for functionality. In future releases, code specific to cloud vendors should be maintained by the cloud vendor themselves, and linked to cloud-controller-manager while running Kubernetes.

The following controllers 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
- Volume Controller: For creating, attaching, and mounting volumes, and interacting with the cloud provider to orchestrate volumes

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 enables the Kubernetes service abstraction by maintaining network rules on the host and performing connection forwarding.

Container Runtime

The container runtime is the software that is responsible for running containers. Kubernetes supports several runtimes: Docker, rkt, runc and any OCI runtime-spec implementation.

Addons

Addons are pods and services that implement cluster features. The pods may be managed by Deployments, ReplicationControllers, and so on. Namespaced addon objects are created in 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.

Edit This Page

The Kubernetes API

Overall API conventions are described in the API conventions doc.

API endpoints, resource types and samples are described in API Reference.

Remote access to the API is discussed in the Controlling API Access doc.

The Kubernetes API also serves as the foundation for the declarative configuration schema for the system. The kubectl command-line tool can be used to create, update, delete, and get API objects.

Kubernetes also stores its serialized state (currently in etcd) in terms of the API resources.

Kubernetes itself is decomposed into multiple components, which interact through its API.

- API changes
- OpenAPI and Swagger definitions
- API versioning
- API groups
- Enabling API groups
- Enabling resources in the groups

API changes

In our experience, any system that is successful needs to grow and change as new use cases emerge or existing ones change. Therefore, we expect the Kubernetes API to continuously change and grow. However, we intend to not break compatibility with existing clients, for an extended period of time. In general, new API resources and new resource fields can be expected to be added frequently. Elimination of resources or fields will require following the API deprecation policy.

What constitutes a compatible change and how to change the API are detailed by the API change document.

OpenAPI and Swagger definitions

Complete API details are documented using Swagger v1.2 and OpenAPI. The Kubernetes apiserver (aka "master") exposes an API that can be used to retrieve the Swagger v1.2 Kubernetes API spec located at `/swaggerapi`.

Starting with Kubernetes 1.10, OpenAPI spec is served in a single `/openapi/v2` endpoint. The format-separated endpoints (`/swagger.json`, `/swagger-2.0.0.json`, `/swagger-2.0.0.pb-v1`, `/swagger-2.0.0.pb-v1.gz`) are deprecated and will get removed in Kubernetes 1.14.

Requested format is specified by setting HTTP headers:

Header	Possible Values
Accept	`application/json`, `application/com.github.proto-openapi.spec.v2@v1.0+protobuf`
Accept-Encoding	`gzip` (not passing this header is acceptable)

Examples of getting OpenAPI spec:

Before 1.10	Starting with Kubernetes 1.10
GET /swagger.json	GET /openapi/v2 **Accept**: application/json
GET /swagger-2.0.0.pb-v1	GET /openapi/v2 **Accept**: application/com.github.proto-openapi.spec.v2@v1
GET /swagger-2.0.0.pb-v1.gz	GET /openapi/v2 **Accept**: application/com.github.proto-openapi.spec.v2@v1

Kubernetes implements an alternative Protobuf based serialization format for the API that is primarily intended for intra-cluster communication, documented in the design proposal and the IDL files for each schema are located in the Go packages that define the API objects.

API 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/extensions/v1beta1`.

We chose to version 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-lifed and/or experimental APIs. The JSON and Protobuf serialization schemas follow the same guidelines for schema changes - all descriptions below cover both formats.

Note that API versioning and Software versioning are only indirectly related. The API and release versioning proposal describes the relationship between API versioning and software versioning.

Different API versions imply different levels of stability and support. The criteria for each level are described in more detail in the API Changes documentation. They are summarized here:

- Alpha level:
  - The version names contain `alpha` (e.g. `v1alpha1`).
  - May be buggy. Enabling the feature may expose bugs. Disabled by default.
  - Support for feature may be dropped at any time without notice.
  - The API may change in incompatible ways in a later software release without notice.
  - Recommended for use only in short-lived testing clusters, due to increased risk of bugs and lack of long-term support.
- Beta level:
  - The version names contain `beta` (e.g. `v2beta3`).
  - Code is well tested. Enabling the feature is considered safe. Enabled by default.
  - Support for the overall feature will not be dropped, though details may change.
  - The schema and/or semantics of objects may change in incompatible ways in a subsequent beta or stable release. When this happens, we will provide instructions for migrating to the next version. This may require deleting, editing, and re-creating API objects. The editing process may require some thought. This may require downtime for applications that rely on the feature.

  - Recommended for only non-business-critical uses because of potential for incompatible changes in subsequent releases. If you have multiple clusters which can be upgraded independently, you may be able to relax this restriction.
  - **Please do try our beta features and give feedback on them! Once they exit beta, it may not be practical for us to make more changes.**
- Stable level:
  - The version name is `vX` where `X` is an integer.
  - Stable versions of features will appear in released software for many subsequent versions.

API groups

To make it easier to extend the Kubernetes API, we implemented *API groups*. The API group is specified in a REST path and in the `apiVersion` field of a serialized object.

Currently there are several API groups in use:

1. The *core* group, often referred to as the *legacy group*, is at the REST path `/api/v1` and uses `apiVersion: v1`.
2. The named groups are at REST path `/apis/$GROUP_NAME/$VERSION`, and use `apiVersion: $GROUP_NAME/$VERSION` (e.g. `apiVersion: batch/v1`). Full list of supported API groups can be seen in Kubernetes API reference.

There are two supported paths to extending the API with custom resources:

1. CustomResourceDefinition is for users with very basic CRUD needs.
2. Users needing the full set of Kubernetes API semantics can implement their own apiserver and use the aggregator to make it seamless for clients.

Enabling API groups

Certain resources and API groups are enabled by default. They can be enabled or disabled by setting `--runtime-config` on apiserver. `--runtime-config` accepts comma separated values. For ex: to disable batch/v1, set `--runtime-config=batch/v1=false`, to enable batch/v2alpha1, set `--runtime-config=batch/v2alpha1`. The flag accepts comma separated set of key=value pairs describing runtime configuration of the apiserver.

IMPORTANT: Enabling or disabling groups or resources requires restarting apiserver and controller-manager to pick up the `--runtime-config` changes.

Enabling resources in the groups

DaemonSets, Deployments, HorizontalPodAutoscalers, Ingress, Jobs and ReplicaSets are enabled by default. Other extensions resources can be enabled by setting `--runtime-config` on apiserver. `--runtime-config` accepts comma separated values. For example: to disable deployments and ingress, set `--runtime-config=extensions/v1beta1/deployments=false,extensions/v1beta1/ingress=false`

Edit This Page

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
- What's next

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

Every Kubernetes object includes two nested object fields that govern the object's configuration: the object *spec* and the object *status*. The *spec*, which you must provide, describes your *desired state* for the object–the characteristics that

you want the object to have. The *status* describes the *actual state* of the object, and is supplied and updated by the Kubernetes system. At any given time, the Kubernetes Control Plane actively manages an object's actual state to match the desired state you supplied.

For example, a Kubernetes 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:

application/deployment.yaml

```
apiVersion: apps/v1 # for versions before 1.9.0 use apps/v1beta2
kind: Deployment
metadata:
  name: nginx-deployment
spec:
  selector:
    matchLabels:
      app: nginx
  replicas: 2 # tells deployment to run 2 pods matching the template
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
      - name: nginx
        image: nginx:1.7.9
        ports:
        - containerPort: 80
```

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

```
$ kubectl create -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`

You'll also need to provide the object `spec` field. 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` object can be found here, and the `spec` format for a `Deployment` object can be found here.

What's next

- Learn about the most important basic Kubernetes objects, such as Pod.

Edit This Page

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
- What's next

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

Every Kubernetes object includes two nested object fields that govern the object's configuration: the object *spec* and the object *status*. The *spec*, which you must provide, describes your *desired state* for the object–the characteristics that you want the object to have. The *status* describes the *actual state* of the object, and is supplied and updated by the Kubernetes system. At any given time, the Kubernetes Control Plane actively manages an object's actual state to match the desired state you supplied.

For example, a Kubernetes 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:

application/deployment.yaml

```
apiVersion: apps/v1 # for versions before 1.9.0 use apps/v1beta2
kind: Deployment
metadata:
  name: nginx-deployment
spec:
  selector:
    matchLabels:
      app: nginx
  replicas: 2 # tells deployment to run 2 pods matching the template
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
      - name: nginx
        image: nginx:1.7.9
        ports:
        - containerPort: 80
```

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

```
$ kubectl create -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`

You'll also need to provide the object `spec` field. 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` object can be found here, and the `spec` format for a `Deployment` object can be found here.

What's next

- Learn about the most important basic Kubernetes objects, such as Pod.

Edit This Page

Names

All objects in the Kubernetes REST API are unambiguously identified by a Name and a UID.

For non-unique user-provided attributes, Kubernetes provides labels and annotations.

See the identifiers design doc for the precise syntax rules for Names and UIDs.

- Names
- UIDs

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.

By convention, the names of Kubernetes resources should be up to maximum length of 253 characters and consist of lower case alphanumeric characters, `-`, and `.`, but certain resources have more specific restrictions.

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.

Edit This Page

Namespaces

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

- When to Use Multiple Namespaces
- Working with Namespaces
- Namespaces and DNS
- Not All Objects are in a Namespace

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

In future versions of Kubernetes, objects in the same namespace will have the same access control policies by default.

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.

Viewing namespaces

You can list the current namespaces in a cluster using:

```
$ kubectl get namespaces
NAME          STATUS    AGE
default       Active    1d
kube-system   Active    1d
kube-public   Active    1d
```

Kubernetes starts with three initial namespaces:

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

Setting the namespace for a request

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

For example:

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

Setting the namespace preference

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

```
$ kubectl config set-context $(kubectl config current-context) --namespace=<insert-namespace
# Validate it
$ kubectl config view | 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
```

Edit This Page

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"
  }
}
```

We'll eventually index and reverse-index labels for efficient queries and watches, use them to sort and group in UIs and CLIs, etc. We don't want to pollute labels with non-identifying, especially large and/or structured, data. Non-identifying information should be recorded using annotations.

- Motivation
- Syntax and character set
- Label selectors
- API

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 represen-

tations, 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/` prefix is 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.

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.

An empty label selector (that is, one with zero requirements) selects every object in the collection.

A null label selector (which is only possible for optional selector fields) selects no objects.

> **Note:** The label selectors of two controllers must not overlap within a namespace, otherwise they will fight with each other.

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+%28fr`

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`, `Replica Set`, and `Daemon Set`, 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.

Edit This Page

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
- What's next

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.

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.

What's next

Learn more about Labels and Selectors.

Edit This Page

Field Selectors

- – Supported fields
  – Supported operators
  – Chained selectors
  – Multiple resource types

Field selectors let you select Kubernetes resources based on the value of one or more resource fields. Here are some example 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 following `kubectl` queries equivalent:

```
$ kubectl get pods
$ kubectl get pods --field-selector ""
```

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 "", fie
```

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 --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 --field-selector metadata.namespace!=default
```

Edit This Page

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.

- Labels
- Applications And Instances Of Applications
- Examples

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
`app.kubernetes.io/name`	The name of the application
`app.kubernetes.io/instance`	A unique name identifying the instance of an application
`app.kubernetes.io/version`	The current version of the application (e.g., a semantic version, revision h
`app.kubernetes.io/component`	The component within the architecture
`app.kubernetes.io/part-of`	The name of a higher level application this one is part of
`app.kubernetes.io/managed-by`	The tool being used to manage the operation of an application

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: wordpress-abcxzy
    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-abcxzy`. 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: wordpress-abcxzy
    app.kubernetes.io/managed-by: helm
    app.kubernetes.io/component: database
    app.kubernetes.io/part-of: wordpress
    app.kubernetes.io/version: "5.7.21"
...
```

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: wordpress-abcxzy
    app.kubernetes.io/managed-by: helm
    app.kubernetes.io/component: database
    app.kubernetes.io/part-of: wordpress
    app.kubernetes.io/version: "5.7.21"
...
```

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

Edit This Page

Managing Kubernetes Objects Using Imperative Commands

Kubernetes objects can quickly be created, updated, and deleted directly using imperative commands built into the `kubectl` command-line tool. This document explains how those commands are organized and how to use them to manage live objects.

- Trade-offs
- How to create objects
- How to update objects
- How to delete objects
- How to view an object
- Using `set` commands to modify objects before creation
- Using `--edit` to modify objects before creation
- What's next

Trade-offs

The `kubectl` tool supports three kinds of object management:

- Imperative commands
- Imperative object configuration
- Declarative object configuration

See Kubernetes Object Management for a discussion of the advantages and disadvantage of each kind of object management.

How to create objects

The `kubectl` tool supports verb-driven commands for creating some of the most common object types. The commands are named to be recognizable to users unfamiliar with the Kubernetes object types.

- `run`: Create a new Deployment object to run Containers in one or more Pods.
- `expose`: Create a new Service object to load balance traffic across Pods.
- `autoscale`: Create a new Autoscaler object to automatically horizontally scale a controller, such as a Deployment.

The `kubectl` tool also supports creation commands driven by object type. These commands support more object types and are more explicit about their intent, but require users to know the type of objects they intend to create.

- `create <objecttype> [<subtype>] <instancename>`

Some objects types have subtypes that you can specify in the `create` command. For example, the Service object has several subtypes including ClusterIP, LoadBalancer, and NodePort. Here's an example that creates a Service with subtype NodePort:

```
kubectl create service nodeport <myservicename>
```

In the preceding example, the `create service nodeport` command is called a subcommand of the `create service` command.

You can use the `-h` flag to find the arguments and flags supported by a subcommand:

```
kubectl create service nodeport -h
```

How to update objects

The `kubectl` command supports verb-driven commands for some common update operations. These commands are named to enable users unfamiliar with Kubernetes objects to perform updates without knowing the specific fields that must be set:

- `scale`: Horizontally scale a controller to add or remove Pods by updating the replica count of the controller.
- `annotate`: Add or remove an annotation from an object.
- `label`: Add or remove a label from an object.

The `kubectl` command also supports update commands driven by an aspect of the object. Setting this aspect may set different fields for different object types:

- `set` : Set an aspect of an object.

Note: In Kubernetes version 1.5, not every verb-driven command has an associated aspect-driven command.

The `kubectl` tool supports these additional ways to update a live object directly, however they require a better understanding of the Kubernetes object schema.

- `edit`: Directly edit the raw configuration of a live object by opening its configuration in an editor.
- `patch`: Directly modify specific fields of a live object by using a patch string. For more details on patch strings, see the patch section in API Conventions.

How to delete objects

You can use the `delete` command to delete an object from a cluster:

- `delete <type>/<name>`

  Note: You can use `kubectl delete` for both imperative commands and imperative object configuration. The difference is in the arguments passed to the command. To use `kubectl delete` as an imperative command, pass the object to be deleted as an argument. Here's an example that passes a Deployment object named nginx:

```
kubectl delete deployment/nginx
```

How to view an object

There are several commands for printing information about an object:

- `get`: Prints basic information about matching objects. Use `get -h` to see a list of options.
- `describe`: Prints aggregated detailed information about matching objects.
- `logs`: Prints the stdout and stderr for a container running in a Pod.

Using `set` commands to modify objects before creation

There are some object fields that don't have a flag you can use in a `create` command. In some of those cases, you can use a combination of `set` and `create` to specify a value for the field before object creation. This is done by piping the output of the `create` command to the `set` command, and then back to the `create` command. Here's an example:

```
kubectl create service clusterip my-svc --clusterip="None" -o yaml --dry-run | kubectl set s
```

1. The `kubectl create service -o yaml --dry-run` command creates the configuration for the Service, but prints it to stdout as YAML instead of sending it to the Kubernetes API server.
2. The `kubectl set selector --local -f - -o yaml` command reads the configuration from stdin, and writes the updated configuration to stdout as YAML.
3. The `kubectl create -f -` command creates the object using the configuration provided via stdin.

Using `--edit` to modify objects before creation

You can use `kubectl create --edit` to make arbitrary changes to an object before it is created. Here's an example:

```
kubectl create service clusterip my-svc --clusterip="None" -o yaml --dry-run > /tmp/srv.yaml
kubectl create --edit -f /tmp/srv.yaml
```

1. The `kubectl create service` command creates the configuration for the Service and saves it to `/tmp/srv.yaml`.
2. The `kubectl create --edit` command opens the configuration file for editing before it creates the object.

What's next

- Managing Kubernetes Objects Using Object Configuration (Imperative)
- Managing Kubernetes Objects Using Object Configuration (Declarative)
- Kubectl Command Reference
- Kubernetes API Reference

Edit This Page

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.

- Management techniques
- Imperative commands
- Imperative object configuration
- Declarative object configuration
- What's next

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	Lea
Imperative commands	Live objects	Development projects	1+	Lov
Imperative object configuration	Individual files	Production projects	1	Mo
Declarative object configuration	Directories of files	Production projects	1+	Hig

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 run nginx --image nginx
```

Do the same thing using a different syntax:

```
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:

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

Recursively process directories:

```
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)
- Kubectl Command Reference
- Kubernetes API Reference

Edit This Page

Managing Kubernetes Objects Using Imperative Commands

Kubernetes objects can quickly be created, updated, and deleted directly using imperative commands built into the `kubectl` command-line tool. This document explains how those commands are organized and how to use them to manage live objects.

- Trade-offs
- How to create objects
- How to update objects
- How to delete objects
- How to view an object
- Using `set` commands to modify objects before creation
- Using `--edit` to modify objects before creation
- What's next

Trade-offs

The `kubectl` tool supports three kinds of object management:

- Imperative commands
- Imperative object configuration
- Declarative object configuration

See Kubernetes Object Management for a discussion of the advantages and disadvantage of each kind of object management.

How to create objects

The `kubectl` tool supports verb-driven commands for creating some of the most common object types. The commands are named to be recognizable to users unfamiliar with the Kubernetes object types.

- `run`: Create a new Deployment object to run Containers in one or more Pods.
- `expose`: Create a new Service object to load balance traffic across Pods.
- `autoscale`: Create a new Autoscaler object to automatically horizontally scale a controller, such as a Deployment.

The `kubectl` tool also supports creation commands driven by object type. These commands support more object types and are more explicit about their intent, but require users to know the type of objects they intend to create.

- `create <objecttype> [<subtype>] <instancename>`

Some objects types have subtypes that you can specify in the `create` command. For example, the Service object has several subtypes including ClusterIP, LoadBalancer, and NodePort. Here's an example that creates a Service with subtype NodePort:

```
kubectl create service nodeport <myservicename>
```

In the preceding example, the `create service nodeport` command is called a subcommand of the `create service` command.

You can use the `-h` flag to find the arguments and flags supported by a subcommand:

```
kubectl create service nodeport -h
```

How to update objects

The `kubectl` command supports verb-driven commands for some common update operations. These commands are named to enable users unfamiliar with Kubernetes objects to perform updates without knowing the specific fields that must be set:

- `scale`: Horizontally scale a controller to add or remove Pods by updating the replica count of the controller.
- `annotate`: Add or remove an annotation from an object.
- `label`: Add or remove a label from an object.

The `kubectl` command also supports update commands driven by an aspect of the object. Setting this aspect may set different fields for different object types:

- `set` : Set an aspect of an object.

Note: In Kubernetes version 1.5, not every verb-driven command has an associated aspect-driven command.

The `kubectl` tool supports these additional ways to update a live object directly, however they require a better understanding of the Kubernetes object schema.

- `edit`: Directly edit the raw configuration of a live object by opening its configuration in an editor.
- `patch`: Directly modify specific fields of a live object by using a patch string. For more details on patch strings, see the patch section in API Conventions.

How to delete objects

You can use the `delete` command to delete an object from a cluster:

- `delete <type>/<name>`

  Note: You can use `kubectl delete` for both imperative commands and imperative object configuration. The difference is in the arguments passed to the command. To use `kubectl delete` as an imperative command, pass the object to be deleted as an argument. Here's an example that passes a Deployment object named nginx:

```
kubectl delete deployment/nginx
```

How to view an object

There are several commands for printing information about an object:

- `get`: Prints basic information about matching objects. Use `get -h` to see a list of options.
- `describe`: Prints aggregated detailed information about matching objects.
- `logs`: Prints the stdout and stderr for a container running in a Pod.

Using `set` commands to modify objects before creation

There are some object fields that don't have a flag you can use in a `create` command. In some of those cases, you can use a combination of `set` and `create` to specify a value for the field before object creation. This is done by piping the output of the `create` command to the `set` command, and then back to the `create` command. Here's an example:

```
kubectl create service clusterip my-svc --clusterip="None" -o yaml --dry-run | kubectl set s
```

1. The `kubectl create service -o yaml --dry-run` command creates the configuration for the Service, but prints it to stdout as YAML instead of sending it to the Kubernetes API server.
2. The `kubectl set selector --local -f - -o yaml` command reads the configuration from stdin, and writes the updated configuration to stdout as YAML.
3. The `kubectl create -f -` command creates the object using the configuration provided via stdin.

Using `--edit` to modify objects before creation

You can use `kubectl create --edit` to make arbitrary changes to an object before it is created. Here's an example:

```
kubectl create service clusterip my-svc --clusterip="None" -o yaml --dry-run > /tmp/srv.yaml
kubectl create --edit -f /tmp/srv.yaml
```

1. The `kubectl create service` command creates the configuration for the Service and saves it to `/tmp/srv.yaml`.
2. The `kubectl create --edit` command opens the configuration file for editing before it creates the object.

What's next

- Managing Kubernetes Objects Using Object Configuration (Imperative)
- Managing Kubernetes Objects Using Object Configuration (Declarative)
- Kubectl Command Reference
- Kubernetes API Reference

Edit This Page

Imperative Management of Kubernetes Objects Using Configuration Files

Kubernetes objects can be created, updated, and deleted by using the `kubectl` command-line tool along with an object configuration file written in YAML or JSON. This document explains how to define and manage objects using configuration files.

- Trade-offs
- How to create objects
- How to update objects
- How to delete objects
- How to view an object

- Limitations
- Creating and editing an object from a URL without saving the configuration
- Migrating from imperative commands to imperative object configuration
- Defining controller selectors and PodTemplate labels
- What's next

Trade-offs

The `kubectl` tool supports three kinds of object management:

- Imperative commands
- Imperative object configuration
- Declarative object configuration

See Kubernetes Object Management for a discussion of the advantages and disadvantage of each kind of object management.

How to create objects

You can use `kubectl create -f` to create an object from a configuration file. Refer to the kubernetes API reference for details.

- `kubectl create -f <filename|url>`

How to update objects

> **Warning:** Updating objects with the `replace` command drops all parts of the spec not specified in the configuration file. This should not be used with objects whose specs are partially managed by the cluster, such as Services of type `LoadBalancer`, where the `externalIPs` field is managed independently from the configuration file. Independently managed fields must be copied to the configuration file to prevent `replace` from dropping them.

You can use `kubectl replace -f` to update a live object according to a configuration file.

- `kubectl replace -f <filename|url>`

How to delete objects

You can use `kubectl delete -f` to delete an object that is described in a configuration file.

- `kubectl delete -f <filename|url>`

How to view an object

You can use `kubectl get -f` to view information about an object that is described in a configuration file.

- `kubectl get -f <filename|url> -o yaml`

The `-o yaml` flag specifies that the full object configuration is printed. Use `kubectl get -h` to see a list of options.

Limitations

The `create`, `replace`, and `delete` commands work well when each object's configuration is fully defined and recorded in its configuration file. However when a live object is updated, and the updates are not merged into its configuration file, the updates will be lost the next time a `replace` is executed. This can happen if a controller, such as a HorizontalPodAutoscaler, makes updates directly to a live object. Here's an example:

1. You create an object from a configuration file.
2. Another source updates the object by changing some field.
3. You replace the object from the configuration file. Changes made by the other source in step 2 are lost.

If you need to support multiple writers to the same object, you can use `kubectl apply` to manage the object.

Creating and editing an object from a URL without saving the configuration

Suppose you have the URL of an object configuration file. You can use `kubectl create --edit` to make changes to the configuration before the object is created. This is particularly useful for tutorials and tasks that point to a configuration file that could be modified by the reader.

```
kubectl create -f <url> --edit
```

Migrating from imperative commands to imperative object configuration

Migrating from imperative commands to imperative object configuration involves several manual steps.

1. Export the live object to a local object configuration file:

   ```
   kubectl get <kind>/<name> -o yaml --export > <kind>_<name>.yaml
   ```

2. Manually remove the status field from the object configuration file.
3. For subsequent object management, use `replace` exclusively.

   ```
   kubectl replace -f <kind>_<name>.yaml
   ```

Defining controller selectors and PodTemplate labels

> **Warning:** Updating selectors on controllers is strongly discouraged.

The recommended approach is to define a single, immutable PodTemplate label used only by the controller selector with no other semantic meaning.

Example label:

```
selector:
  matchLabels:
      controller-selector: "extensions/v1beta1/deployment/nginx"
template:
  metadata:
    labels:
      controller-selector: "extensions/v1beta1/deployment/nginx"
```

What's next

- Managing Kubernetes Objects Using Imperative Commands
- Managing Kubernetes Objects Using Object Configuration (Declarative)
- Kubectl Command Reference
- Kubernetes API Reference

Edit This Page

Declarative Management of Kubernetes Objects Using Configuration Files

Kubernetes objects can be created, updated, and deleted by storing multiple object configuration files in a directory and using `kubectl apply` to recursively create and update those objects as needed. This method retains writes made to live objects without merging the changes back into the object configuration files.

- Trade-offs

- Before you begin
- How to create objects
- How to update objects
- How to delete objects
- How to view an object
- How apply calculates differences and merges changes
- Default field values
- How to change ownership of a field between the configuration file and direct imperative writers
- Changing management methods
- Defining controller selectors and PodTemplate labels
- What's next

Trade-offs

The `kubectl` tool supports three kinds of object management:

- Imperative commands
- Imperative object configuration
- Declarative object configuration

See Kubernetes Object Management for a discussion of the advantages and disadvantage of each kind of object management.

Before you begin

Declarative object configuration requires a firm understanding of the Kubernetes object definitions and configuration. Read and complete the following documents if you have not already:

- Managing Kubernetes Objects Using Imperative Commands
- Imperative Management of Kubernetes Objects Using Configuration Files

Following are definitions for terms used in this document:

- *object configuration file / configuration file*: A file that defines the configuration for a Kubernetes object. This topic shows how to pass configuration files to `kubectl apply`. Configuration files are typically stored in source control, such as Git.
- *live object configuration / live configuration*: The live configuration values of an object, as observed by the Kubernetes cluster. These are kept in the Kubernetes cluster storage, typically etcd.
- *declarative configuration writer / declarative writer*: A person or software component that makes updates to a live object. The live writers referred to in this topic make changes to object configuration files and run `kubectl apply` to write the changes.

How to create objects

Use `kubectl apply` to create all objects, except those that already exist, defined by configuration files in a specified directory:

```
kubectl apply -f <directory>/
```

This sets the `kubectl.kubernetes.io/last-applied-configuration: '{...}'` annotation on each object. The annotation contains the contents of the object configuration file that was used to create the object.

Note: Add the `-R` flag to recursively process directories.

Here's an example of an object configuration file:

application/simple_deployment.yaml

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-deployment
spec:
  selector:
    matchLabels:
      app: nginx
  minReadySeconds: 5
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
      - name: nginx
        image: nginx:1.7.9
        ports:
        - containerPort: 80
```

Create the object using `kubectl apply`:

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

Print the live configuration using `kubectl get`:

```
kubectl get -f https://k8s.io/examples/application/simple_deployment.yaml -o yaml
```

The output shows that the `kubectl.kubernetes.io/last-applied-configuration`

annotation was written to the live configuration, and it matches the configuration file:

```
kind: Deployment
metadata:
  annotations:
    # ...
    # This is the json representation of simple_deployment.yaml
    # It was written by kubectl apply when the object was created
    kubectl.kubernetes.io/last-applied-configuration: |
      {"apiVersion":"apps/v1","kind":"Deployment",
      "metadata":{"annotations":{},"name":"nginx-deployment","namespace":"default"},
      "spec":{"minReadySeconds":5,"selector":{"matchLabels":{"app":nginx}},"template":{"meta
      "spec":{"containers":[{"image":"nginx:1.7.9","name":"nginx",
      "ports":[{"containerPort":80}]}]}}}}
  # ...
spec:
  # ...
  minReadySeconds: 5
  selector:
    matchLabels:
      # ...
      app: nginx
  template:
    metadata:
      # ...
      labels:
        app: nginx
    spec:
      containers:
      - image: nginx:1.7.9
        # ...
        name: nginx
        ports:
        - containerPort: 80
        # ...
      # ...
    # ...
  # ...
```

How to update objects

You can also use `kubectl apply` to update all objects defined in a directory, even if those objects already exist. This approach accomplishes the following:

1. Sets fields that appear in the configuration file in the live configuration.

2. Clears fields removed from the configuration file in the live configuration.

```
kubectl apply -f <directory>/
```

Note: Add the `-R` flag to recursively process directories.

Here's an example configuration file:

application/simple_deployment.yaml

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-deployment
spec:
  selector:
    matchLabels:
      app: nginx
  minReadySeconds: 5
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
      - name: nginx
        image: nginx:1.7.9
        ports:
        - containerPort: 80
```

Create the object using `kubectl apply`:

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

Note: For purposes of illustration, the preceding command refers to a single configuration file instead of a directory.

Print the live configuration using `kubectl get`:

```
kubectl get -f https://k8s.io/examples/application/simple_deployment.yaml -o yaml
```

The output shows that the `kubectl.kubernetes.io/last-applied-configuration` annotation was written to the live configuration, and it matches the configuration file:

```
kind: Deployment
metadata:
```

```
  annotations:
    # ...
    # This is the json representation of simple_deployment.yaml
    # It was written by kubectl apply when the object was created
    kubectl.kubernetes.io/last-applied-configuration: |
      {"apiVersion":"apps/v1","kind":"Deployment",
      "metadata":{"annotations":{},"name":"nginx-deployment","namespace":"default"},
      "spec":{"minReadySeconds":5,"selector":{"matchLabels":{"app":nginx}},"template":{"meta
      "spec":{"containers":[{"image":"nginx:1.7.9","name":"nginx",
      "ports":[{"containerPort":80}]}]}}}}
  # ...
spec:
  # ...
  minReadySeconds: 5
  selector:
    matchLabels:
      # ...
      app: nginx
  template:
    metadata:
      # ...
      labels:
        app: nginx
    spec:
      containers:
      - image: nginx:1.7.9
        # ...
        name: nginx
        ports:
        - containerPort: 80
        # ...
      # ...
    # ...
  # ...
```

Directly update the `replicas` field in the live configuration by using `kubectl scale`. This does not use `kubectl apply`:

```
kubectl scale deployment/nginx-deployment --replicas=2
```

Print the live configuration using `kubectl get`:

```
kubectl get -f https://k8s.io/examples/application/simple_deployment.yaml -o yaml
```

The output shows that the `replicas` field has been set to 2, and the `last-applied-configuration` annotation does not contain a `replicas` field:

```
apiVersion: apps/v1
kind: Deployment
```

```
metadata:
  annotations:
    # ...
    # note that the annotation does not contain replicas
    # because it was not updated through apply
    kubectl.kubernetes.io/last-applied-configuration: |
      {"apiVersion":"apps/v1","kind":"Deployment",
      "metadata":{"annotations":{},"name":"nginx-deployment","namespace":"default"},
      "spec":{"minReadySeconds":5,"selector":{"matchLabels":{"app":nginx}},"template":{"meta
      "spec":{"containers":[{"image":"nginx:1.7.9","name":"nginx",
      "ports":[{"containerPort":80}]}]}}}}
  # ...
spec:
  replicas: 2 # written by scale
  # ...
  minReadySeconds: 5
  selector:
    matchLabels:
      # ...
      app: nginx
  template:
    metadata:
      # ...
      labels:
        app: nginx
    spec:
      containers:
      - image: nginx:1.7.9
        # ...
        name: nginx
        ports:
        - containerPort: 80
      # ...
```

Update the `simple_deployment.yaml` configuration file to change the image from `nginx:1.7.9` to `nginx:1.11.9`, and delete the `minReadySeconds` field:

application/update_deployment.yaml

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-deployment
spec:
  selector:
    matchLabels:
      app: nginx
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
      - name: nginx
        image: nginx:1.11.9 # update the image
        ports:
        - containerPort: 80
```

Apply the changes made to the configuration file:

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

Print the live configuration using `kubectl get`:

```
kubectl get -f https://k8s.io/examples/application/simple_deployment.yaml -o yaml
```

The output shows the following changes to the live configuration:

- The `replicas` field retains the value of 2 set by `kubectl scale`. This is possible because it is omitted from the configuration file.
- The `image` field has been updated to `nginx:1.11.9` from `nginx:1.7.9`.
- The `last-applied-configuration` annotation has been updated with the new image.
- The `minReadySeconds` field has been cleared.
- The `last-applied-configuration` annotation no longer contains the `minReadySeconds` field.

```
apiVersion: apps/v1
kind: Deployment
metadata:
  annotations:
    # ...
    # The annotation contains the updated image to nginx 1.11.9,
```

```
    # but does not contain the updated replicas to 2
    kubectl.kubernetes.io/last-applied-configuration: |
      {"apiVersion":"apps/v1","kind":"Deployment",
      "metadata":{"annotations":{},"name":"nginx-deployment","namespace":"default"},
      "spec":{"selector":{"matchLabels":{"app":nginx}},"template":{"metadata":{"labels":{"ap
      "spec":{"containers":[{"image":"nginx:1.11.9","name":"nginx",
      "ports":[{"containerPort":80}]}]}}}}
    # ...
spec:
  replicas: 2 # Set by `kubectl scale`.  Ignored by `kubectl apply`.
  # minReadySeconds cleared by `kubectl apply`
  # ...
  selector:
    matchLabels:
      # ...
      app: nginx
  template:
    metadata:
      # ...
      labels:
        app: nginx
    spec:
      containers:
      - image: nginx:1.11.9 # Set by `kubectl apply`
        # ...
        name: nginx
        ports:
        - containerPort: 80
        # ...
      # ...
    # ...
  # ...
```

Warning: Mixing `kubectl apply` with the imperative object configuration commands `create` and `replace` is not supported. This is because `create` and `replace` do not retain the `kubectl.kubernetes.io/last-applied-configuration` that `kubectl apply` uses to compute updates.

How to delete objects

There are two approaches to delete objects managed by `kubectl apply`.

Recommended: `kubectl delete -f <filename>`

Manually deleting objects using the imperative command is the recommended approach, as it is more explicit about what is being deleted, and less likely to result in the user deleting something unintentionally:

```
kubectl delete -f <filename>
```

Alternative: `kubectl apply -f <directory/> --prune -l your=label`

Only use this if you know what you are doing.

> **Warning:** `kubectl apply --prune` is in alpha, and backwards incompatible changes might be introduced in subsequent releases.
>
> **Warning:** You must be careful when using this command, so that you do not delete objects unintentionally.

As an alternative to `kubectl delete`, you can use `kubectl apply` to identify objects to be deleted after their configuration files have been removed from the directory. Apply with `--prune` queries the API server for all objects matching a set of labels, and attempts to match the returned live object configurations against the object configuration files. If an object matches the query, and it does not have a configuration file in the directory, and it has a `last-applied-configuration` annotation, it is deleted.

```
kubectl apply -f <directory/> --prune -l <labels>
```

> **Warning:** Apply with prune should only be run against the root directory containing the object configuration files. Running against sub-directories can cause objects to be unintentionally deleted if they are returned by the label selector query specified with `-l <labels>` and do not appear in the subdirectory.

How to view an object

You can use `kubectl get` with `-o yaml` to view the configuration of a live object:

```
kubectl get -f <filename|url> -o yaml
```

How apply calculates differences and merges changes

> **Caution:** A *patch* is an update operation that is scoped to specific fields of an object instead of the entire object. This enables updating only a specific set of fields on an object without reading the object first.

When `kubectl apply` updates the live configuration for an object, it does so by sending a patch request to the API server. The patch defines updates scoped to specific fields of the live object configuration. The `kubectl apply` command calculates this patch request using the configuration file, the live configuration, and the `last-applied-configuration` annotation stored in the live configuration.

Merge patch calculation

The `kubectl apply` command writes the contents of the configuration file to the `kubectl.kubernetes.io/last-applied-configuration` annotation. This is used to identify fields that have been removed from the configuration file and need to be cleared from the live configuration. Here are the steps used to calculate which fields should be deleted or set:

1. Calculate the fields to delete. These are the fields present in `last-applied-configuration` and missing from the configuration file.
2. Calculate the fields to add or set. These are the fields present in the configuration file whose values don't match the live configuration.

Here's an example. Suppose this is the configuration file for a Deployment object:

application/update_deployment.yaml

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-deployment
spec:
  selector:
    matchLabels:
      app: nginx
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
      - name: nginx
        image: nginx:1.11.9 # update the image
        ports:
        - containerPort: 80
```

Also, suppose this is the live configuration for the same Deployment object:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  annotations:
    # ...
    # note that the annotation does not contain replicas
    # because it was not updated through apply
    kubectl.kubernetes.io/last-applied-configuration: |
      {"apiVersion":"apps/v1","kind":"Deployment",
      "metadata":{"annotations":{},"name":"nginx-deployment","namespace":"default"},
      "spec":{"minReadySeconds":5,"selector":{"matchLabels":{"app":nginx}},"template":{"meta
      "spec":{"containers":[{"image":"nginx:1.7.9","name":"nginx",
      "ports":[{"containerPort":80}]}]}}}}
  # ...
spec:
  replicas: 2 # written by scale
  # ...
  minReadySeconds: 5
  selector:
    matchLabels:
      # ...
      app: nginx
  template:
    metadata:
      # ...
      labels:
        app: nginx
    spec:
      containers:
      - image: nginx:1.7.9
        # ...
        name: nginx
        ports:
        - containerPort: 80
      # ...
```

Here are the merge calculations that would be performed by `kubectl apply`:

1. Calculate the fields to delete by reading values from `last-applied-configuration` and comparing them to values in the configuration file. Clear fields explicitly set to null in the local object configuration file regardless of whether they appear in the `last-applied-configuration`. In this example, `minReadySeconds` appears in the `last-applied-configuration` annotation, but does not appear in the configuration file. **Action:** Clear `minReadySeconds` from the live configuration.

2. Calculate the fields to set by reading values from the configuration file and comparing them to values in the live configuration. In this example, the value of `image` in the configuration file does not match the value in the live configuration. **Action:** Set the value of `image` in the live configuration.
3. Set the `last-applied-configuration` annotation to match the value of the configuration file.
4. Merge the results from 1, 2, 3 into a single patch request to the API server.

Here is the live configuration that is the result of the merge:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  annotations:
    # ...
    # The annotation contains the updated image to nginx 1.11.9,
    # but does not contain the updated replicas to 2
    kubectl.kubernetes.io/last-applied-configuration: |
      {"apiVersion":"apps/v1","kind":"Deployment",
      "metadata":{"annotations":{},"name":"nginx-deployment","namespace":"default"},
      "spec":{"selector":{"matchLabels":{"app":nginx}},"template":{"metadata":{"labels":{"ap
      "spec":{"containers":[{"image":"nginx:1.11.9","name":"nginx",
      "ports":[{"containerPort":80}]}]}}}}
    # ...
spec:
  selector:
    matchLabels:
      # ...
      app: nginx
  replicas: 2 # Set by `kubectl scale`.  Ignored by `kubectl apply`.
  # minReadySeconds cleared by `kubectl apply`
  # ...
  template:
    metadata:
      # ...
      labels:
        app: nginx
    spec:
      containers:
      - image: nginx:1.11.9 # Set by `kubectl apply`
        # ...
        name: nginx
        ports:
        - containerPort: 80
        # ...
      # ...
    # ...
```

```
# ...
```

How different types of fields are merged

How a particular field in a configuration file is merged with the live configuration depends on the type of the field. There are several types of fields:

- *primitive*: A field of type string, integer, or boolean. For example, `image` and `replicas` are primitive fields. **Action:** Replace.
- *map*, also called *object*: A field of type map or a complex type that contains subfields. For example, `labels`, `annotations`,`spec` and `metadata` are all maps. **Action:** Merge elements or subfields.
- *list*: A field containing a list of items that can be either primitive types or maps. For example, `containers`, `ports`, and `args` are lists. **Action:** Varies.

When `kubectl apply` updates a map or list field, it typically does not replace the entire field, but instead updates the individual subelements. For instance, when merging the `spec` on a Deployment, the entire `spec` is not replaced. Instead the subfields of `spec`, such as `replicas`, are compared and merged.

Merging changes to primitive fields

Primitive fields are replaced or cleared.

Note: **-** is used for "not applicable" because the value is not used.

Field in object configuration file	Field in live object configuration	Field in last-applied-configuration	Ac
Yes	Yes	-	Set
Yes	No	-	Set
No	-	Yes	Cle
No	-	No	Do

Merging changes to map fields

Fields that represent maps are merged by comparing each of the subfields or elements of the map:

Note: **-** is used for "not applicable" because the value is not used.

Key in object configuration file	Key in live object configuration	Field in last-applied-configuration	Actio
Yes	Yes	-	Comp
Yes	No	-	Set li

Key in object configuration file	Key in live object configuration	Field in last-applied-configuration	Actio
No	-	Yes	Delet
No	-	No	Do n

Merging changes for fields of type list

Merging changes to a list uses one of three strategies:

- Replace the list.
- Merge individual elements in a list of complex elements.
- Merge a list of primitive elements.

The choice of strategy is made on a per-field basis.

Replace the list

Treat the list the same as a primitive field. Replace or delete the entire list. This preserves ordering.

Example: Use `kubectl apply` to update the `args` field of a Container in a Pod. This sets the value of `args` in the live configuration to the value in the configuration file. Any `args` elements that had previously been added to the live configuration are lost. The order of the `args` elements defined in the configuration file is retained in the live configuration.

```
# last-applied-configuration value
    args: ["a", "b"]

# configuration file value
    args: ["a", "c"]

# live configuration
    args: ["a", "b", "d"]

# result after merge
    args: ["a", "c"]
```

Explanation: The merge used the configuration file value as the new list value.

Merge individual elements of a list of complex elements:

Treat the list as a map, and treat a specific field of each element as a key. Add, delete, or update individual elements. This does not preserve ordering.

This merge strategy uses a special tag on each field called a `patchMergeKey`. The `patchMergeKey` is defined for each field in the Kubernetes source code:

types.go When merging a list of maps, the field specified as the `patchMergeKey` for a given element is used like a map key for that element.

Example: Use `kubectl apply` to update the `containers` field of a PodSpec. This merges the list as though it was a map where each element is keyed by `name`.

```
# last-applied-configuration value
    containers:
    - name: nginx
      image: nginx:1.10
    - name: nginx-helper-a # key: nginx-helper-a; will be deleted in result
      image: helper:1.3
    - name: nginx-helper-b # key: nginx-helper-b; will be retained
      image: helper:1.3

# configuration file value
    containers:
    - name: nginx
      image: nginx:1.10
    - name: nginx-helper-b
      image: helper:1.3
    - name: nginx-helper-c # key: nginx-helper-c; will be added in result
      image: helper:1.3

# live configuration
    containers:
    - name: nginx
      image: nginx:1.10
    - name: nginx-helper-a
      image: helper:1.3
    - name: nginx-helper-b
      image: helper:1.3
      args: ["run"] # Field will be retained
    - name: nginx-helper-d # key: nginx-helper-d; will be retained
      image: helper:1.3

# result after merge
    containers:
    - name: nginx
      image: nginx:1.10
      # Element nginx-helper-a was deleted
    - name: nginx-helper-b
      image: helper:1.3
      args: ["run"] # Field was retained
    - name: nginx-helper-c # Element was added
      image: helper:1.3
```

```
- name: nginx-helper-d # Element was ignored
  image: helper:1.3
```

Explanation:

- The container named "nginx-helper-a" was deleted because no container named "nginx-helper-a" appeared in the configuration file.
- The container named "nginx-helper-b" retained the changes to `args` in the live configuration. `kubectl apply` was able to identify that "nginx-helper-b" in the live configuration was the same "nginx-helper-b" as in the configuration file, even though their fields had different values (no `args` in the configuration file). This is because the `patchMergeKey` field value (name) was identical in both.
- The container named "nginx-helper-c" was added because no container with that name appeared in the live configuration, but one with that name appeared in the configuration file.
- The container named "nginx-helper-d" was retained because no element with that name appeared in the last-applied-configuration.

Merge a list of primitive elements

As of Kubernetes 1.5, merging lists of primitive elements is not supported.

> **Note:** Which of the above strategies is chosen for a given field is controlled by the `patchStrategy` tag in types.go If no `patchStrategy` is specified for a field of type list, then the list is replaced.

Default field values

The API server sets certain fields to default values in the live configuration if they are not specified when the object is created.

Here's a configuration file for a Deployment. The file does not specify `strategy`:

application/simple_deployment.yaml

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-deployment
spec:
  selector:
    matchLabels:
      app: nginx
  minReadySeconds: 5
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
      - name: nginx
        image: nginx:1.7.9
        ports:
        - containerPort: 80
```

Create the object using `kubectl apply`:

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

Print the live configuration using `kubectl get`:

```
kubectl get -f https://k8s.io/examples/application/simple_deployment.yaml -o yaml
```

The output shows that the API server set several fields to default values in the live configuration. These fields were not specified in the configuration file.

```
apiVersion: apps/v1
kind: Deployment
# ...
spec:
  selector:
    matchLabels:
      app: nginx
  minReadySeconds: 5
  replicas: 1 # defaulted by apiserver
  strategy:
    rollingUpdate: # defaulted by apiserver - derived from strategy.type
      maxSurge: 1
```

```
      maxUnavailable: 1
    type: RollingUpdate # defaulted apiserver
  template:
    metadata:
      creationTimestamp: null
      labels:
        app: nginx
    spec:
      containers:
      - image: nginx:1.7.9
        imagePullPolicy: IfNotPresent # defaulted by apiserver
        name: nginx
        ports:
        - containerPort: 80
          protocol: TCP # defaulted by apiserver
        resources: {} # defaulted by apiserver
        terminationMessagePath: /dev/termination-log # defaulted by apiserver
      dnsPolicy: ClusterFirst # defaulted by apiserver
      restartPolicy: Always # defaulted by apiserver
      securityContext: {} # defaulted by apiserver
      terminationGracePeriodSeconds: 30 # defaulted by apiserver
# ...
```

In a patch request, defaulted fields are not re-defaulted unless they are explicitly cleared as part of a patch request. This can cause unexpected behavior for fields that are defaulted based on the values of other fields. When the other fields are later changed, the values defaulted from them will not be updated unless they are explicitly cleared.

For this reason, it is recommended that certain fields defaulted by the server are explicitly defined in the configuration file, even if the desired values match the server defaults. This makes it easier to recognize conflicting values that will not be re-defaulted by the server.

Example:

```
# last-applied-configuration
spec:
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
      - name: nginx
        image: nginx:1.7.9
        ports:
        - containerPort: 80
```

```
# configuration file
spec:
  strategy:
    type: Recreate # updated value
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
      - name: nginx
        image: nginx:1.7.9
        ports:
        - containerPort: 80

# live configuration
spec:
  strategy:
    type: RollingUpdate # defaulted value
    rollingUpdate: # defaulted value derived from type
      maxSurge : 1
      maxUnavailable: 1
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
      - name: nginx
        image: nginx:1.7.9
        ports:
        - containerPort: 80

# result after merge - ERROR!
spec:
  strategy:
    type: Recreate # updated value: incompatible with rollingUpdate
    rollingUpdate: # defaulted value: incompatible with "type: Recreate"
      maxSurge : 1
      maxUnavailable: 1
  template:
    metadata:
      labels:
        app: nginx
    spec:
```

```
containers:
- name: nginx
  image: nginx:1.7.9
  ports:
  - containerPort: 80
```

Explanation:

1. The user creates a Deployment without defining `strategy.type`.
2. The server defaults `strategy.type` to `RollingUpdate` and defaults the `strategy.rollingUpdate` values.
3. The user changes `strategy.type` to `Recreate`. The `strategy.rollingUpdate` values remain at their defaulted values, though the server expects them to be cleared. If the `strategy.rollingUpdate` values had been defined initially in the configuration file, it would have been more clear that they needed to be deleted.
4. Apply fails because `strategy.rollingUpdate` is not cleared. The `strategy.rollingupdate` field cannot be defined with a `strategy.type` of `Recreate`.

Recommendation: These fields should be explicitly defined in the object configuration file:

- Selectors and PodTemplate labels on workloads, such as Deployment, StatefulSet, Job, DaemonSet, ReplicaSet, and ReplicationController
- Deployment rollout strategy

How to clear server-defaulted fields or fields set by other writers

Fields that do not appear in the configuration file can be cleared by setting their values to `null` and then applying the configuration file. For fields defaulted by the server, this triggers re-defaulting the values.

How to change ownership of a field between the configuration file and direct imperative writers

These are the only methods you should use to change an individual object field:

- Use `kubectl apply`.
- Write directly to the live configuration without modifying the configuration file: for example, use `kubectl scale`.

Changing the owner from a direct imperative writer to a configuration file

Add the field to the configuration file. For the field, discontinue direct updates to the live configuration that do not go through `kubectl apply`.

Changing the owner from a configuration file to a direct imperative writer

As of Kubernetes 1.5, changing ownership of a field from a configuration file to an imperative writer requires manual steps:

- Remove the field from the configuration file.
- Remove the field from the `kubectl.kubernetes.io/last-applied-configuration` annotation on the live object.

Changing management methods

Kubernetes objects should be managed using only one method at a time. Switching from one method to another is possible, but is a manual process.

> **Note:** It is OK to use imperative deletion with declarative management.

Migrating from imperative command management to declarative object configuration

Migrating from imperative command management to declarative object configuration involves several manual steps:

1. Export the live object to a local configuration file:

   ```
   kubectl get <kind>/<name> -o yaml --export > <kind>_<name>.yaml
   ```

2. Manually remove the `status` field from the configuration file.

   > **Note:**
   >
   > `This step is optional, as `kubectl apply` does not update the status field even if it is present in the configuration file.`

3. Set the `kubectl.kubernetes.io/last-applied-configuration` annotation on the object:

   ```
   kubectl replace --save-config -f <kind>_<name>.yaml
   ```

4. Change processes to use `kubectl apply` for managing the object exclusively.

Migrating from imperative object configuration to declarative object configuration

1. Set the `kubectl.kubernetes.io/last-applied-configuration` annotation on the object:

   ```
   kubectl replace --save-config -f <kind>_<name>.yaml
   ```

2. Change processes to use `kubectl apply` for managing the object exclusively.

Defining controller selectors and PodTemplate labels

> **Warning:** Updating selectors on controllers is strongly discouraged.

The recommended approach is to define a single, immutable PodTemplate label used only by the controller selector with no other semantic meaning.

Example:

```
selector:
  matchLabels:
      controller-selector: "extensions/v1beta1/deployment/nginx"
template:
  metadata:
    labels:
      controller-selector: "extensions/v1beta1/deployment/nginx"
```

What's next

- Managing Kubernetes Objects Using Imperative Commands
- Imperative Management of Kubernetes Objects Using Configuration Files
- Kubectl Command Reference
- Kubernetes API Reference

Edit This Page

Cluster Networking

Kubernetes approaches networking somewhat differently than Docker does by default. There are 4 distinct networking problems to solve:

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.

- Docker model
- Kubernetes model
- How to implement the Kubernetes networking model
- What's next

Kubernetes assumes that pods can communicate with other pods, regardless of which host they land on. Every pod gets its own IP address so 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.

There are requirements imposed on how you set up your cluster networking to achieve this.

Docker model

Before discussing the Kubernetes approach to networking, it is worthwhile to review the "normal" way that networking works with Docker. By default, Docker uses host-private networking. It creates a virtual bridge, called `docker0` by default, and allocates a subnet from one of the private address blocks defined in RFC1918 for that bridge. For each container that Docker creates, it allocates a virtual Ethernet device (called `veth`) which is attached to the bridge. The veth is mapped to appear as `eth0` in the container, using Linux namespaces. The in-container `eth0` interface is given an IP address from the bridge's address range.

The result is that Docker containers can talk to other containers only if they are on the same machine (and thus the same virtual bridge). Containers on different machines can not reach each other - in fact they may end up with the exact same network ranges and IP addresses.

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.

Kubernetes model

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.

Kubernetes imposes the following fundamental requirements on any networking implementation (barring any intentional network segmentation policies):

- all containers can communicate with all other containers without NAT
- all nodes can communicate with all containers (and vice-versa) without NAT
- the IP that a container sees itself as is the same IP that others see it as

What this means in practice is that you can not just take two computers running Docker and expect Kubernetes to work. You must ensure that the fundamental requirements are met.

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.

Until now this document has talked about containers. In reality, Kubernetes applies IP addresses at the `Pod` scope - containers within a `Pod` share their network namespaces - including their IP address. This means that containers within a `Pod` can all reach each other's ports on `localhost`. This does imply that containers within a `Pod` must coordinate port usage, but this is no different than processes in a VM. This is called the "IP-per-pod" model. This is implemented, using Docker, as a "pod container" which holds the network namespace open while "app containers" (the things the user specified) join that namespace with Docker's `--net=container:<id>` function.

As with Docker, it is possible to request host ports, but this is reduced to a very niche operation. In this case a port will be allocated on the host `Node` and traffic will be forwarded to the `Pod`. 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.

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.

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: http://www.apstra.com/products/how-it-works/

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 along side 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.

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.

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.

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

Contrail, based on OpenContrail, is a truly open, multi-cloud network virtualization and policy management platform. Contrail / OpenContrail is integrated with various orchestration systems such as Kubernetes, OpenShift, OpenStack and Mesos, and provides 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 `cbr0`) 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.

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-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.

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.

Project Calico

Project Calico is an open source container networking provider and network policy engine.

Calico provides a highly scalable networking and network policy solution for connecting Kubernetes pods based on the same IP networking principles as the internet. Calico can be deployed without encapsulation or overlays to provide high-performance, high-scale data center networking. Calico also provides fine-grained, intent based network security policy for Kubernetes pods via its distributed firewall.

Calico can also be run in policy enforcement mode in conjunction with other networking solutions such as Flannel, aka canal, or native GCE networking.

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.

Edit This Page

Cluster Administration Overview

The cluster administration overview is for anyone creating or administering a Kubernetes cluster. It assumes some familiarity with core Kubernetes concepts.

- Planning a cluster
- Managing a cluster
- Securing a cluster
- Optional Cluster Services

Planning a cluster

See the guides in Picking the Right Solution for examples of how to plan, set up, and configure Kubernetes clusters. The solutions listed in this article are called *distros*.

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.
- **If you are designing for high-availability**, learn about configuring clusters in multiple zones.
- 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.

Note: Not all distros are actively maintained. Choose distros which have been tested with a recent version of Kubernetes.

Managing a cluster

- Managing a cluster describes several topics related to the lifecycle of a cluster: creating a new cluster, upgrading your cluster's master and worker

nodes, performing node maintenance (e.g. kernel upgrades), and upgrading the Kubernetes API version of a running cluster.

- 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 to set up permissions for users and service accounts.
- 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

- Master-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.

Edit This Page

Certificates

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

- Distributing Self-Signed CA Certificate
- Certificates API

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 CA. (`--batch` set automatic mode. `--req-cn` default CN to use.)

   ```
   ./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 assume that you are using `cluster.local` as the default DNS domain name.

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

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

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

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

openssl

openssl can manually generate certificates for your cluster.

1. Generate a ca.key with 2048bit:

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

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

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

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 assume 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://pkg.cfssl.org/R1.2/cfssl_linux-amd64 -o cfssl
   chmod +x cfssl
   curl -L https://pkg.cfssl.org/R1.2/cfssljson_linux-amd64 -o cfssljson
   chmod +x cfssljson
   curl -L https://pkg.cfssl.org/R1.2/cfssl-certinfo_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 the 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 as shown below. 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 assume 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.

Edit This Page

Cloud Providers

This page explains how to manage Kubernetes running on a specific cloud provider.

- AWS
- Azure
- CloudStack
- GCE
- OpenStack
- OVirt
- Photon
- VSphere
- IBM Cloud Kubernetes Service

kubeadm

kubeadm is a popular option for creating kubernetes clusters. kubeadm has configuration options to specify configuration information for cloud providers. For example a typical in-tree cloud provider can be configured using kubeadm as shown below:

```
apiVersion: kubeadm.k8s.io/v1alpha3
kind: InitConfiguration
nodeRegistration:
  kubeletExtraArgs:
    cloud-provider: "openstack"
    cloud-config: "/etc/kubernetes/cloud.conf"
---
kind: ClusterConfiguration
apiVersion: kubeadm.k8s.io/v1alpha3
```

```
kubernetesVersion: v1.12.0
apiServerExtraArgs:
  cloud-provider: "openstack"
  cloud-config: "/etc/kubernetes/cloud.conf"
apiServerExtraVolumes:
- name: cloud
  hostPath: "/etc/kubernetes/cloud.conf"
  mountPath: "/etc/kubernetes/cloud.conf"
controllerManagerExtraArgs:
  cloud-provider: "openstack"
  cloud-config: "/etc/kubernetes/cloud.conf"
controllerManagerExtraVolumes:
- name: cloud
  hostPath: "/etc/kubernetes/cloud.conf"
  mountPath: "/etc/kubernetes/cloud.conf"
```

The in-tree cloud providers typically need both `--cloud-provider` and `--cloud-config` specified in the command lines for the kube-apiserver, kube-controller-manager and the kubelet. The contents of the file specified in `--cloud-config` for each provider is documented below as well.

For all external cloud providers, please follow the instructions on the individual repositories.

AWS

This section describes all the possible configurations which can be used when running Kubernetes on Amazon Web Services.

Node Name

The AWS cloud provider uses the private DNS name of the AWS instance as the name of the Kubernetes Node object.

Load Balancers

You can setup external load balancers to use specific features in AWS by configuring the annotations as shown below.

```
apiVersion: v1
kind: Service
metadata:
  name: example
  namespace: kube-system
  labels:
```

```
    run: example
  annotations:
     service.beta.kubernetes.io/aws-load-balancer-ssl-cert: arn:aws:acm:xx-xxxx-x:xxxxxxxxx
     service.beta.kubernetes.io/aws-load-balancer-backend-protocol: http
spec:
  type: LoadBalancer
  ports:
  - port: 443
    targetPort: 5556
    protocol: TCP
  selector:
    app: example
```

Different settings can be applied to a load balancer service in AWS using *annotations*. The following describes the annotations supported on AWS ELBs:

- `service.beta.kubernetes.io/aws-load-balancer-access-log-emit-interval`: Used to specify access log emit interval.
- `service.beta.kubernetes.io/aws-load-balancer-access-log-enabled`: Used on the service to enable or disable access logs.
- `service.beta.kubernetes.io/aws-load-balancer-access-log-s3-bucket-name`: Used to specify access log s3 bucket name.
- `service.beta.kubernetes.io/aws-load-balancer-access-log-s3-bucket-prefix`: Used to specify access log s3 bucket prefix.
- `service.beta.kubernetes.io/aws-load-balancer-additional-resource-tags`: Used on the service to specify a comma-separated list of key-value pairs which will be recorded as additional tags in the ELB. For example: `"Key1=Val1,Key2=Val2,KeyNoVal1=,KeyNoVal2"`.
- `service.beta.kubernetes.io/aws-load-balancer-backend-protocol`: Used on the service to specify the protocol spoken by the backend (pod) behind a listener. If `http` (default) or `https`, an HTTPS listener that terminates the connection and parses headers is created. If set to `ssl` or `tcp`, a "raw" SSL listener is used. If set to `http` and `aws-load-balancer-ssl-cert` is not used then a HTTP listener is used.
- `service.beta.kubernetes.io/aws-load-balancer-ssl-cert`: Used on the service to request a secure listener. Value is a valid certificate ARN. For more, see ELB Listener Config CertARN is an IAM or CM certificate ARN, e.g. `arn:aws:acm:us-east-1:123456789012:certificate/12345678-1234-1234-1234-123456789012`
- `service.beta.kubernetes.io/aws-load-balancer-connection-draining-enabled`: Used on the service to enable or disable connection draining.
- `service.beta.kubernetes.io/aws-load-balancer-connection-draining-timeout`: Used on the service to specify a connection draining timeout.
- `service.beta.kubernetes.io/aws-load-balancer-connection-idle-timeout`: Used on the service to specify the idle connection timeout.
- `service.beta.kubernetes.io/aws-load-balancer-cross-zone-load-balancing-enabled`: Used on the service to enable or disable cross-zone load balancing.
- `service.beta.kubernetes.io/aws-load-balancer-extra-security-groups`:

Used on the service to specify additional security groups to be added to ELB created

- `service.beta.kubernetes.io/aws-load-balancer-internal`: Used on the service to indicate that we want an internal ELB.
- `service.beta.kubernetes.io/aws-load-balancer-proxy-protocol`: Used on the service to enable the proxy protocol on an ELB. Right now we only accept the value `*` which means enable the proxy protocol on all ELB backends. In the future we could adjust this to allow setting the proxy protocol only on certain backends.
- `service.beta.kubernetes.io/aws-load-balancer-ssl-ports`: Used on the service to specify a comma-separated list of ports that will use SSL/HTTPS listeners. Defaults to `*` (all)

The information for the annotations for AWS is taken from the comments on aws.go

Azure

Node Name

The Azure cloud provider uses the hostname of the node (as determined by the kubelet or overridden with `--hostname-override`) as the name of the Kubernetes Node object. Note that the Kubernetes Node name must match the Azure VM name.

CloudStack

Node Name

The CloudStack cloud provider uses the hostname of the node (as determined by the kubelet or overridden with `--hostname-override`) as the name of the Kubernetes Node object. Note that the Kubernetes Node name must match the CloudStack VM name.

GCE

Node Name

The GCE cloud provider uses the hostname of the node (as determined by the kubelet or overridden with `--hostname-override`) as the name of the Kubernetes Node object. Note that the first segment of the Kubernetes Node name must match the GCE instance name (e.g. a Node named

`kubernetes-node-2.c.my-proj.internal` must correspond to an instance named `kubernetes-node-2`).

OpenStack

This section describes all the possible configurations which can be used when using OpenStack with Kubernetes.

Node Name

The OpenStack cloud provider uses the instance name (as determined from OpenStack metadata) as the name of the Kubernetes Node object. Note that the instance name must be a valid Kubernetes Node name in order for the kubelet to successfully register its Node object.

Services

The OpenStack cloud provider implementation for Kubernetes supports the use of these OpenStack services from the underlying cloud, where available:

Service	API Version(s)	Required
Block Storage (Cinder)	V1†, V2, V3	No
Compute (Nova)	V2	No
Identity (Keystone)	V2‡, V3	Yes
Load Balancing (Neutron)	V1§, V2	No
Load Balancing (Octavia)	V2	No

† Block Storage V1 API support is deprecated, Block Storage V3 API support was added in Kubernetes 1.9.

‡ Identity V2 API support is deprecated and will be removed from the provider in a future release. As of the "Queens" release, OpenStack will no longer expose the Identity V2 API.

§ Load Balancing V1 API support was removed in Kubernetes 1.9.

Service discovery is achieved by listing the service catalog managed by OpenStack Identity (Keystone) using the `auth-url` provided in the provider configuration. The provider will gracefully degrade in functionality when OpenStack services other than Keystone are not available and simply disclaim support for impacted features. Certain features are also enabled or disabled based on the list of extensions published by Neutron in the underlying cloud.

cloud.conf

Kubernetes knows how to interact with OpenStack via the file cloud.conf. It is the file that will provide Kubernetes with credentials and location for the OpenStack auth endpoint. You can create a cloud.conf file by specifying the following details in it

Typical configuration

This is an example of a typical configuration that touches the values that most often need to be set. It points the provider at the OpenStack cloud's Keystone endpoint, provides details for how to authenticate with it, and configures the load balancer:

```
[Global]
username=user
password=pass
auth-url=https://<keystone_ip>/identity/v3
tenant-id=c869168a828847f39f7f06edd7305637
domain-id=2a73b8f597c04551a0fdc8e95544be8a

[LoadBalancer]
subnet-id=6937f8fa-858d-4bc9-a3a5-18d2c957166a
```

Global

These configuration options for the OpenStack provider pertain to its global configuration and should appear in the `[Global]` section of the `cloud.conf` file:

- `auth-url` (Required): The URL of the keystone API used to authenticate. On OpenStack control panels, this can be found at Access and Security > API Access > Credentials.
- `username` (Required): Refers to the username of a valid user set in keystone.
- `password` (Required): Refers to the password of a valid user set in keystone.
- `tenant-id` (Required): Used to specify the id of the project where you want to create your resources.
- `tenant-name` (Optional): Used to specify the name of the project where you want to create your resources.
- `trust-id` (Optional): Used to specify the identifier of the trust to use for authorization. A trust represents a user's (the trustor) authorization to delegate roles to another user (the trustee), and optionally allow the trustee to impersonate the trustor. Available trusts are found under the `/v3/OS-TRUST/trusts` endpoint of the Keystone API.

- `domain-id` (Optional): Used to specify the id of the domain your user belongs to.
- `domain-name` (Optional): Used to specify the name of the domain your user belongs to.
- `region` (Optional): Used to specify the identifier of the region to use when running on a multi-region OpenStack cloud. A region is a general division of an OpenStack deployment. Although a region does not have a strict geographical connotation, a deployment can use a geographical name for a region identifier such as `us-east`. Available regions are found under the `/v3/regions` endpoint of the Keystone API.
- `ca-file` (Optional): Used to specify the path to your custom CA file.

When using Keystone V3 - which changes tenant to project - the `tenant-id` value is automatically mapped to the project construct in the API.

Load Balancer

These configuration options for the OpenStack provider pertain to the load balancer and should appear in the `[LoadBalancer]` section of the `cloud.conf` file:

- `lb-version` (Optional): Used to override automatic version detection. Valid values are `v1` or `v2`. Where no value is provided automatic detection will select the highest supported version exposed by the underlying OpenStack cloud.
- `use-octavia` (Optional): Used to determine whether to look for and use an Octavia LBaaS V2 service catalog endpoint. Valid values are `true` or `false`. Where `true` is specified and an Octaiva LBaaS V2 entry can not be found, the provider will fall back and attempt to find a Neutron LBaaS V2 endpoint instead. The default value is `false`.
- `subnet-id` (Optional): Used to specify the id of the subnet you want to create your loadbalancer on. Can be found at Network > Networks. Click on the respective network to get its subnets.
- `floating-network-id` (Optional): If specified, will create a floating IP for the load balancer.
- `lb-method` (Optional): Used to specify algorithm by which load will be distributed amongst members of the load balancer pool. The value can be `ROUND_ROBIN`, `LEAST_CONNECTIONS`, or `SOURCE_IP`. The default behavior if none is specified is `ROUND_ROBIN`.
- `lb-provider` (Optional): Used to specify the provider of the load balancer. If not specified, the default provider service configured in neutron will be used.
- `create-monitor` (Optional): Indicates whether or not to create a health monitor for the Neutron load balancer. Valid values are `true` and `false`. The default is `false`. When `true` is specified then `monitor-delay`, `monitor-timeout`, and `monitor-max-retries` must also be set.

- `monitor-delay` (Optional): The time, in seconds, between sending probes to members of the load balancer.
- `monitor-timeout` (Optional): Maximum number of seconds for a monitor to wait for a ping reply before it times out. The value must be less than the delay value.
- `monitor-max-retries` (Optional): Number of permissible ping failures before changing the load balancer member's status to INACTIVE. Must be a number between 1 and 10.
- `manage-security-groups` (Optional): Determines whether or not the load balancer should automatically manage the security group rules. Valid values are `true` and `false`. The default is `false`. When `true` is specified `node-security-group` must also be supplied.
- `node-security-group` (Optional): ID of the security group to manage.

Block Storage

These configuration options for the OpenStack provider pertain to block storage and should appear in the `[BlockStorage]` section of the `cloud.conf` file:

- `bs-version` (Optional): Used to override automatic version detection. Valid values are `v1`, `v2`, `v3` and `auto`. When `auto` is specified automatic detection will select the highest supported version exposed by the underlying OpenStack cloud. The default value if none is provided is `auto`.
- `trust-device-path` (Optional): In most scenarios the block device names provided by Cinder (e.g. `/dev/vda`) can not be trusted. This boolean toggles this behavior. Setting it to `true` results in trusting the block device names provided by Cinder. The default value of `false` results in the discovery of the device path based on its serial number and `/dev/disk/by-id` mapping and is the recommended approach.
- `ignore-volume-az` (Optional): Used to influence availability zone use when attaching Cinder volumes. When Nova and Cinder have different availability zones, this should be set to `true`. This is most commonly the case where there are many Nova availability zones but only one Cinder availability zone. The default value is `false` to preserve the behavior used in earlier releases, but may change in the future.

If deploying Kubernetes versions <= 1.8 on an OpenStack deployment that uses paths rather than ports to differentiate between endpoints it may be necessary to explicitly set the `bs-version` parameter. A path based endpoint is of the form `http://foo.bar/volume` while a port based endpoint is of the form `http://foo.bar:xxx`.

In environments that use path based endpoints and Kubernetes is using the older auto-detection logic a `BS API version autodetection failed.` error will be returned on attempting volume detachment. To workaround this issue it is possible to force the use of Cinder API version 2 by adding this to the cloud provider configuration:

```
[BlockStorage]
bs-version=v2
```

Metadata

These configuration options for the OpenStack provider pertain to metadata and should appear in the `[Metadata]` section of the `cloud.conf` file:

- `search-order` (Optional): This configuration key influences the way that the provider retrieves metadata relating to the instance(s) in which it runs. The default value of `configDrive,metadataService` results in the provider retrieving metadata relating to the instance from the config drive first if available and then the metadata service. Alternative values are:
  - `configDrive` - Only retrieve instance metadata from the configuration drive.
  - `metadataService` - Only retrieve instance metadata from the metadata service.
  - `metadataService,configDrive` - Retrieve instance metadata from the metadata service first if available, then the configuration drive.

Influencing this behavior may be desirable as the metadata on the configuration drive may grow stale over time, whereas the metadata service always provides the most up to date view. Not all OpenStack clouds provide both configuration drive and metadata service though and only one or the other may be available which is why the default is to check both.

Router

These configuration options for the OpenStack provider pertain to the kubenet Kubernetes network plugin and should appear in the `[Router]` section of the `cloud.conf` file:

- `router-id` (Optional): If the underlying cloud's Neutron deployment supports the `extraroutes` extension then use `router-id` to specify a router to add routes to. The router chosen must span the private networks containing your cluster nodes (typically there is only one node network, and this value should be the default router for the node network). This value is required to use kubenet on OpenStack.

OVirt

Node Name

The OVirt cloud provider uses the hostname of the node (as determined by the kubelet or overridden with `--hostname-override`) as the name of the Kubernetes Node object. Note that the Kubernetes

Node name must match the VM FQDN (reported by OVirt under `<vm><guest_info><fqdn>...</fqdn></guest_info></vm>`)

Photon

Node Name

The Photon cloud provider uses the hostname of the node (as determined by the kubelet or overridden with `--hostname-override`) as the name of the Kubernetes Node object. Note that the Kubernetes Node name must match the Photon VM name (or if `overrideIP` is set to true in the `--cloud-config`, the Kubernetes Node name must match the Photon VM IP address).

VSphere

Node Name

The VSphere cloud provider uses the detected hostname of the node (as determined by the kubelet) as the name of the Kubernetes Node object.

The `--hostname-override` parameter is ignored by the VSphere cloud provider.

IBM Cloud Kubernetes Service

Compute nodes

By using the IBM Cloud Kubernetes Service provider, you can create clusters with a mixture of virtual and physical (bare metal) nodes in a single zone or across multiple zones in a region. For more information, see Planning your cluster and worker node setup.

The name of the Kubernetes Node object is the private IP address of the IBM Cloud Kubernetes Service worker node instance.

Networking

The IBM Cloud Kubernetes Service provider provides VLANs for quality network performance and network isolation for nodes. You can set up custom firewalls and Calico network policies to add an extra layer of security for your cluster, or connect your cluster to your on-prem data center via VPN. For more information, see Planning in-cluster and private networking.

To expose apps to the public or within the cluster, you can leverage NodePort, LoadBalancer, or Ingress services. You can also customize the Ingress application load balancer with annotations. For more information, see Planning to expose your apps with external networking.

Storage

The IBM Cloud Kubernetes Service provider leverages Kubernetes-native persistent volumes to enable users to mount file, block, and cloud object storage to their apps. You can also use database-as-a-service and third-party add-ons for persistent storage of your data. For more information, see Planning highly available persistent storage.

Edit This Page

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
- Bulk operations in kubectl
- Using labels effectively
- Canary deployments
- Updating labels
- Updating annotations
- Scaling your application
- In-place updates of resources
- Disruptive updates
- Updating your application without a service outage
- What's next

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:

application/nginx-app.yaml

```
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:1.7.9
        ports:
        - containerPort: 80
```

Multiple resources can be created the same way as a single resource:

```
$ kubectl create -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 create` also accepts multiple `-f` arguments:

```
$ kubectl create -f https://k8s.io/examples/application/nginx/nginx-svc.yaml -f https://k8s.
```

And a directory can be specified rather than or in addition to individual files:

```
$ kubectl create -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 create -f https://raw.githubusercontent.com/kubernetes/website/master/content/en/e
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
NAME           TYPE           CLUSTER-IP   EXTERNAL-IP   PORT(S)      AGE
my-nginx-svc   LoadBalancer   10.0.0.208   <pending>     80/TCP       0s
```

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 create -f project/k8s/development
error: you must provide one or more resources by argument or filename (.json|.yaml|.yml|stdi
```

Instead, specify the `--recursive` or `-R` flag with the `--filename,-f` flag as such:

```
$ kubectl create -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 create -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 create -f examples/guestbook/all-in-one/guestbook-all-in-one.yaml
$ kubectl get pods -Lapp -Ltier -Lrole
NAME                           READY     STATUS    RESTARTS   AGE       APP         TIER
guestbook-fe-4nlpb             1/1       Running   0          1m        guestbook   frontend
guestbook-fe-ght6d             1/1       Running   0          1m        guestbook   frontend
guestbook-fe-jpy62             1/1       Running   0          1m        guestbook   frontend
guestbook-redis-master-5pg3b   1/1       Running   0          1m        guestbook   backend
guestbook-redis-slave-2q2yf    1/1       Running   0          1m        guestbook   backend
guestbook-redis-slave-qgazl    1/1       Running   0          1m        guestbook   backend
my-nginx-divi2                 1/1       Running   0          29m       nginx       <none>
my-nginx-o0ef1                 1/1       Running   0          29m       nginx       <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.extensions/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.

> **Note:** To use apply, always create resource initially with either `kubectl apply` or `kubectl create --save-config`.

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. If your deployed application is managed by Replication Controllers, you should read how to use `kubectl rolling-update` instead.

Let's say you were running version 1.7.9 of nginx:

```
$ kubectl run my-nginx --image=nginx:1.7.9 --replicas=3
deployment.apps/my-nginx created
```

To update to version 1.9.1, simply change `.spec.template.spec.containers[0].image` from `nginx:1.7.9` to `nginx:1.9.1`, 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.
- Configuration Best Practices and Tips

Edit This Page

Cluster Networking

Kubernetes approaches networking somewhat differently than Docker does by default. There are 4 distinct networking problems to solve:

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.

- Docker model
- Kubernetes model
- How to implement the Kubernetes networking model
- What's next

Kubernetes assumes that pods can communicate with other pods, regardless of which host they land on. Every pod gets its own IP address so 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.

There are requirements imposed on how you set up your cluster networking to achieve this.

Docker model

Before discussing the Kubernetes approach to networking, it is worthwhile to review the "normal" way that networking works with Docker. By default, Docker uses host-private networking. It creates a virtual bridge, called `docker0` by default, and allocates a subnet from one of the private address blocks defined in RFC1918 for that bridge. For each container that Docker creates, it allocates a virtual Ethernet device (called `veth`) which is attached to the bridge. The veth is mapped to appear as `eth0` in the container, using Linux namespaces. The in-container `eth0` interface is given an IP address from the bridge's address range.

The result is that Docker containers can talk to other containers only if they are on the same machine (and thus the same virtual bridge). Containers on different machines can not reach each other - in fact they may end up with the exact same network ranges and IP addresses.

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.

Kubernetes model

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.

Kubernetes imposes the following fundamental requirements on any networking implementation (barring any intentional network segmentation policies):

- all containers can communicate with all other containers without NAT
- all nodes can communicate with all containers (and vice-versa) without NAT
- the IP that a container sees itself as is the same IP that others see it as

What this means in practice is that you can not just take two computers running Docker and expect Kubernetes to work. You must ensure that the fundamental requirements are met.

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.

Until now this document has talked about containers. In reality, Kubernetes applies IP addresses at the `Pod` scope - containers within a `Pod` share their network namespaces - including their IP address. This means that containers within a `Pod` can all reach each other's ports on `localhost`. This does imply that containers within a `Pod` must coordinate port usage, but this is no different than processes in a VM. This is called the "IP-per-pod" model. This is implemented, using Docker, as a "pod container" which holds the network namespace open while "app containers" (the things the user specified) join that namespace with Docker's `--net=container:<id>` function.

As with Docker, it is possible to request host ports, but this is reduced to a very niche operation. In this case a port will be allocated on the host `Node` and traffic will be forwarded to the `Pod`. 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.

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.

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: http://www.apstra.com/products/how-it-works/

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 along side 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.

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.

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.

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

Contrail, based on OpenContrail, is a truly open, multi-cloud network virtualization and policy management platform. Contrail / OpenContrail is integrated with various orchestration systems such as Kubernetes, OpenShift, OpenStack and Mesos, and provides 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 `cbr0`) 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.

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-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.

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.

Project Calico

Project Calico is an open source container networking provider and network policy engine.

Calico provides a highly scalable networking and network policy solution for connecting Kubernetes pods based on the same IP networking principles as the internet. Calico can be deployed without encapsulation or overlays to provide high-performance, high-scale data center networking. Calico also provides fine-grained, intent based network security policy for Kubernetes pods via its distributed firewall.

Calico can also be run in policy enforcement mode in conjunction with other networking solutions such as Flannel, aka canal, or native GCE networking.

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.

Edit This Page

Logging Architecture

Application and systems logs can help you understand what is happening inside your cluster. 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.

- Basic logging in Kubernetes
- Logging at the node level
- Cluster-level logging architectures

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:

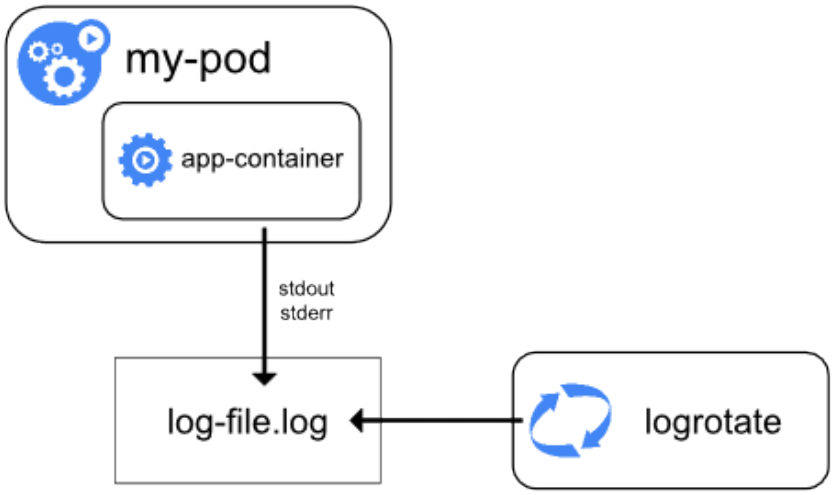
```
$ kubectl create -f https://k8s.io/examples/debug/counter-pod.yaml
pod/counter created
```

To fetch the logs, use the `kubectl logs` command, as follows:

```
$ kubectl logs counter
0: Mon Jan  1 00:00:00 UTC 2001
1: Mon Jan  1 00:00:01 UTC 2001
2: Mon Jan  1 00:00:02 UTC 2001
...
```

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, e.g. 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] [cosConfigure-

Helper].

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 glog logging library. You can find the conventions for logging severity for those components in the development docs on logging.

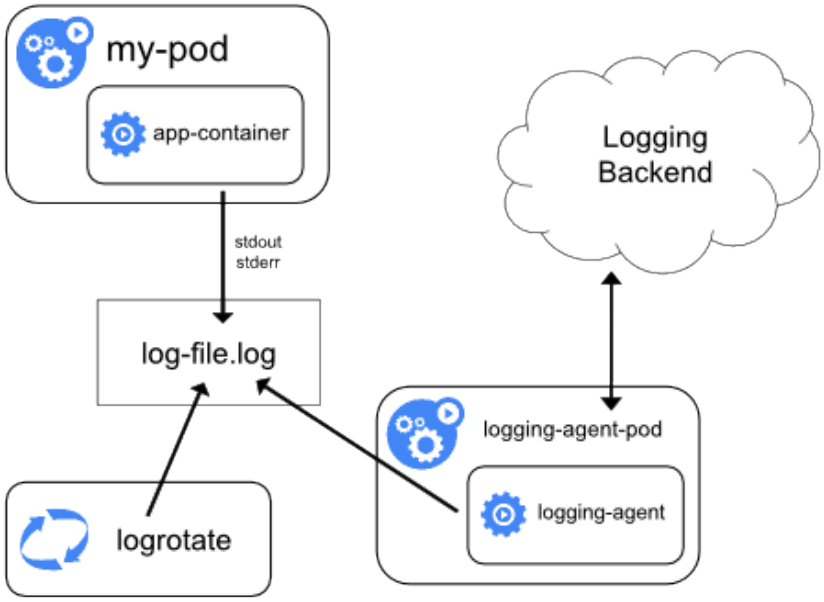
Similarly to the container logs, system component logs in the `/var/log` directory should be rotated. In Kubernetes clusters brought up by the `kube-up.sh` script, those logs are configured to be rotated by the `logrotate` tool daily or once the size exceeds 100MB.

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*.

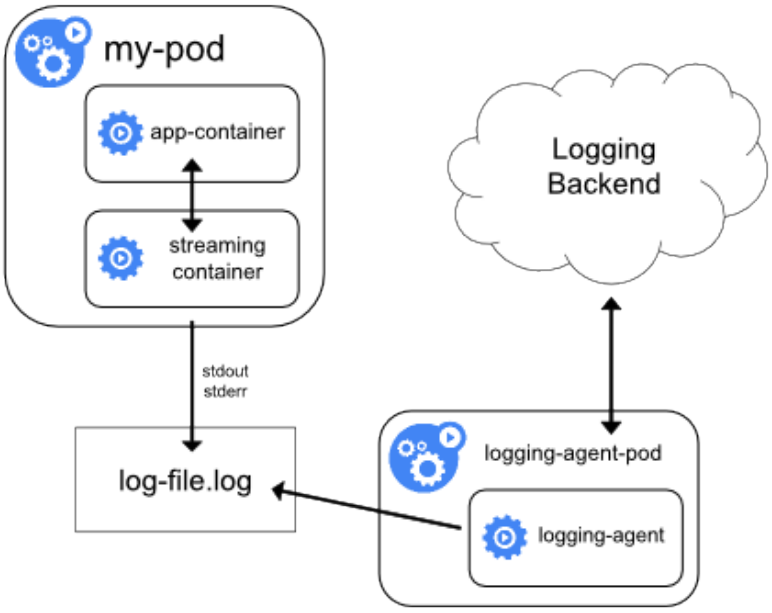
Kubernetes doesn't specify a logging agent, but two optional logging agents are packaged with the Kubernetes release: Stackdriver Logging for use with Google Cloud Platform, and Elasticsearch. You can find more information and instructions in the dedicated documents. Both use fluentd with custom configuration 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

```
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

```
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-log-1
    image: busybox
    args: [/bin/sh, -c, 'tail -n+1 -f /var/log/1.log']
    volumeMounts:
    - name: varlog
      mountPath: /var/log
  - name: count-log-2
    image: busybox
    args: [/bin/sh, -c, 'tail -n+1 -f /var/log/2.log']
    volumeMounts:
    - name: varlog
      mountPath: /var/log
  volumes:
  - name: varlog
    emptyDir: {}
```

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
...
```

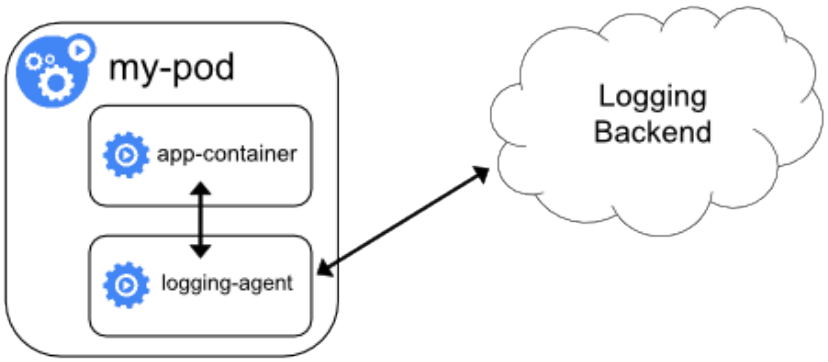
```
$ kubectl logs counter count-log-2
Mon Jan  1 00:00:00 UTC 2001 INFO 0
Mon Jan  1 00:00:01 UTC 2001 INFO 1
Mon Jan  1 00:00:02 UTC 2001 INFO 2
...
```

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.

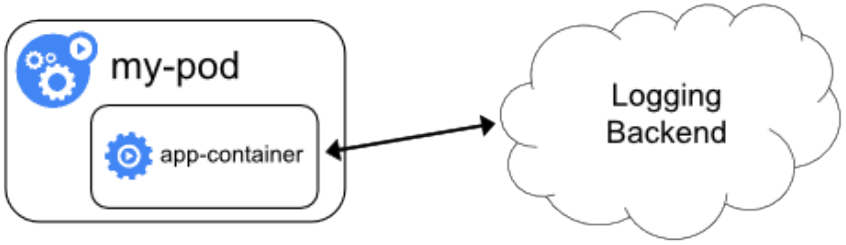
admin/logging/two-files-counter-pod-agent-sidecar.yaml

```
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.

Edit This Page

Configuring kubelet Garbage Collection

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
- Container Collection
- User Configuration
- Deprecation
- What's next

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`). `MaxPerPodContainer` 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

Users 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%.

We also allow users to customize 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 evict
`--image-gc-low-threshold`	`--eviction-minimum-reclaim`	eviction recla
`--maximum-dead-containers`		deprecated o
`--maximum-dead-containers-per-container`		deprecated o
`--minimum-container-ttl-duration`		deprecated o
`--low-diskspace-threshold-mb`	`--eviction-hard` or `eviction-soft`	eviction gene
`--outofdisk-transition-frequency`	`--eviction-pressure-transition-period`	eviction gene

What's next

See Configuring Out Of Resource Handling for more details.

Edit This Page

Federation

`Federation V1`, the current Kubernetes federation API which reuses the Kubernetes API resources 'as is', is currently considered alpha for many of its features. There is no clear path to evolve the API to GA; however, there is a `Federation V2` effort in progress to implement a dedicated federation API apart from the Kubernetes API. The details are available at sig-multicluster community page.

This page explains why and how to manage multiple Kubernetes clusters using federation.

- Why federation
- Setting up federation
- API resources
- Cascading deletion

- Scope of a single cluster
- Selecting the right number of clusters
- What's next

Why federation

Federation makes it easy to manage multiple clusters. It does so by providing 2 major building blocks:

- Sync resources across clusters: Federation provides the ability to keep resources in multiple clusters in sync. For example, you can ensure that the same deployment exists in multiple clusters.
- Cross cluster discovery: Federation provides the ability to auto-configure DNS servers and load balancers with backends from all clusters. For example, you can ensure that a global VIP or DNS record can be used to access backends from multiple clusters.

Some other use cases that federation enables are:

- High Availability: By spreading load across clusters and auto configuring DNS servers and load balancers, federation minimises the impact of cluster failure.
- Avoiding provider lock-in: By making it easier to migrate applications across clusters, federation prevents cluster provider lock-in.

Federation is not helpful unless you have multiple clusters. Some of the reasons why you might want multiple clusters are:

- Low latency: Having clusters in multiple regions minimises latency by serving users from the cluster that is closest to them.
- Fault isolation: It might be better to have multiple small clusters rather than a single large cluster for fault isolation (for example: multiple clusters in different availability zones of a cloud provider).
- Scalability: There are scalability limits to a single kubernetes cluster (this should not be the case for most users. For more details: Kubernetes Scaling and Performance Goals).
- Hybrid cloud: You can have multiple clusters on different cloud providers or on-premises data centers.

Caveats

While there are a lot of attractive use cases for federation, there are also some caveats:

- Increased network bandwidth and cost: The federation control plane watches all clusters to ensure that the current state is as expected. This

can lead to significant network cost if the clusters are running in different regions on a cloud provider or on different cloud providers.
- Reduced cross cluster isolation: A bug in the federation control plane can impact all clusters. This is mitigated by keeping the logic in federation control plane to a minimum. It mostly delegates to the control plane in kubernetes clusters whenever it can. The design and implementation also errs on the side of safety and avoiding multi-cluster outage.
- Maturity: The federation project is relatively new and is not very mature. Not all resources are available and many are still alpha. Issue 88 enumerates known issues with the system that the team is busy solving.

Hybrid cloud capabilities

Federations of Kubernetes Clusters can include clusters running in different cloud providers (e.g. Google Cloud, AWS), and on-premises (e.g. on OpenStack). Kubefed is the recommended way to deploy federated clusters.

Thereafter, your API resources can span different clusters and cloud providers.

Setting up federation

To be able to federate multiple clusters, you first need to set up a federation control plane. Follow the setup guide to set up the federation control plane.

API resources

Once you have the control plane set up, you can start creating federation API resources. The following guides explain some of the resources in detail:

- Cluster
- ConfigMap
- DaemonSets
- Deployment
- Events
- Hpa
- Ingress
- Jobs
- Namespaces
- ReplicaSets
- Secrets
- Services

The API reference docs list all the resources supported by federation apiserver.

Cascading deletion

Kubernetes version 1.6 includes support for cascading deletion of federated resources. With cascading deletion, when you delete a resource from the federation control plane, you also delete the corresponding resources in all underlying clusters.

Cascading deletion is not enabled by default when using the REST API. To enable it, set the option `DeleteOptions.orphanDependents=false` when you delete a resource from the federation control plane using the REST API. Using `kubectl delete` enables cascading deletion by default. You can disable it by running `kubectl delete --cascade=false`

Note: Kubernetes version 1.5 included cascading deletion support for a subset of federation resources.

Scope of a single cluster

On IaaS providers such as Google Compute Engine or Amazon Web Services, a VM exists in a zone or availability zone. We suggest that all the VMs in a Kubernetes cluster should be in the same availability zone, because:

- compared to having a single global Kubernetes cluster, there are fewer single-points of failure.
- compared to a cluster that spans availability zones, it is easier to reason about the availability properties of a single-zone cluster.
- when the Kubernetes developers are designing the system (e.g. making assumptions about latency, bandwidth, or correlated failures) they are assuming all the machines are in a single data center, or otherwise closely connected.

It is recommended to run fewer clusters with more VMs per availability zone; but it is possible to run multiple clusters per availability zones.

Reasons to prefer fewer clusters per availability zone are:

- improved bin packing of Pods in some cases with more nodes in one cluster (less resource fragmentation).
- reduced operational overhead (though the advantage is diminished as ops tooling and processes mature).
- reduced costs for per-cluster fixed resource costs, e.g. apiserver VMs (but small as a percentage of overall cluster cost for medium to large clusters).

Reasons to have multiple clusters include:

- strict security policies requiring isolation of one class of work from another (but, see Partitioning Clusters below).
- test clusters to canary new Kubernetes releases or other cluster software.

Selecting the right number of clusters

The selection of the number of Kubernetes clusters may be a relatively static choice, only revisited occasionally. By contrast, the number of nodes in a cluster and the number of pods in a service may change frequently according to load and growth.

To pick the number of clusters, first, decide which regions you need to be in to have adequate latency to all your end users, for services that will run on Kubernetes (if you use a Content Distribution Network, the latency requirements for the CDN-hosted content need not be considered). Legal issues might influence this as well. For example, a company with a global customer base might decide to have clusters in US, EU, AP, and SA regions. Call the number of regions to be in `R`.

Second, decide how many clusters should be able to be unavailable at the same time, while still being available. Call the number that can be unavailable `U`. If you are not sure, then 1 is a fine choice.

If it is allowable for load-balancing to direct traffic to any region in the event of a cluster failure, then you need at least the larger of `R` or `U + 1` clusters. If it is not (e.g. you want to ensure low latency for all users in the event of a cluster failure), then you need to have `R * (U + 1)` clusters (`U + 1` in each of `R` regions). In any case, try to put each cluster in a different zone.

Finally, if any of your clusters would need more than the maximum recommended number of nodes for a Kubernetes cluster, then you may need even more clusters. Kubernetes v1.3 supports clusters up to 1000 nodes in size. Kubernetes v1.8 supports clusters up to 5000 nodes. See Building Large Clusters for more guidance.

What's next

- Learn more about the Federation proposal.
- See this setup guide for cluster federation.
- See this Kubecon2016 talk on federation
- See this Kubecon2017 Europe update on federation
- See this Kubecon2018 Europe update on sig-multicluster
- See this Kubecon2018 Europe Federation-v2 prototype presentation
- See this Federation-v2 Userguide

Edit This Page

Proxies in Kubernetes

This page explains proxies used with Kubernetes.

- Proxies
- Requesting redirects

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.

Edit This Page

Controller manager metrics

Controller manager metrics provide important insight into the performance and health of the controller manager.

- What are controller manager metrics
- Configuration

What are 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"}
```

Configuration

In a cluster, controller-manager metrics are available from `http://localhost:10252/metrics` from the host where the controller-manager is running.

The metrics are emitted in prometheus format and are human readable.

In a production environment you may want to configure prometheus or some other metrics scraper to periodically gather these metrics and make them available in some kind of time series database.

Edit This Page

Installing Addons

Add-ons extend the functionality of Kubernetes.

This page lists some of the available add-ons and links to their respective installation instructions.

Add-ons in each section are sorted alphabetically - the ordering does not imply any preferential status.

- Networking and Network Policy
- Service Discovery
- Visualization & Control
- Legacy Add-ons

Networking and Network Policy

- ACI provides integrated container networking and network security with Cisco ACI.
- Calico is a secure L3 networking and network policy provider.
- 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.
- 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.
- Flannel is an overlay network provider that can be used with Kubernetes.
- Knitter is a network solution supporting multiple networking in Kubernetes.

- 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.
- 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.

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!

Edit This Page

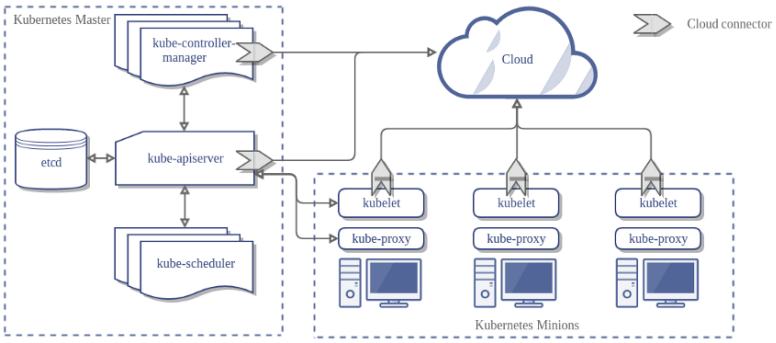
Concepts Underlying the Cloud Controller Manager

The cloud controller manager (CCM) concept (not to be confused with the binary) was originally created to allow cloud specific vendor code and the Kubernetes core to evolve independent of one another. The cloud controller manager runs alongside other master components such as the Kubernetes controller manager, the API server, and scheduler. It can also be started as a Kubernetes addon, in which case it runs on top of Kubernetes.

The cloud controller manager's design is based on a plugin mechanism that allows new cloud providers to integrate with Kubernetes easily by using plugins. There are plans in place for on-boarding new cloud providers on Kubernetes and for migrating cloud providers from the old model to the new CCM model.

This document discusses the concepts behind the cloud controller manager and gives details about its associated functions.

Here's the architecture of a Kubernetes cluster without the cloud controller manager:



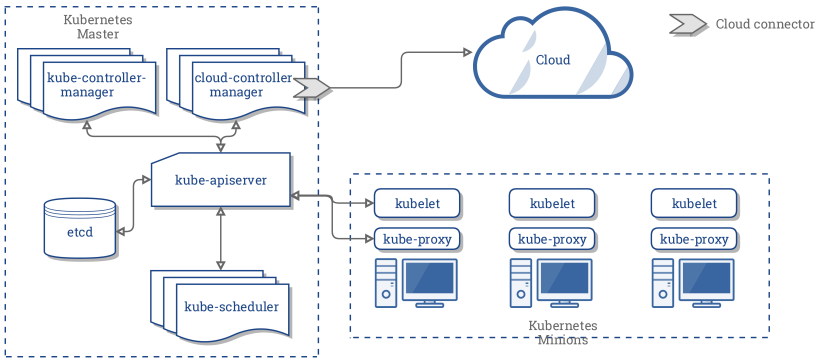
- Design
- Components of the CCM
- Functions of the CCM
- Plugin mechanism
- Authorization
- Vendor Implementations
- Cluster Administration

Design

In the preceding diagram, Kubernetes and the cloud provider are integrated through several different components:

- Kubelet
- Kubernetes controller manager
- Kubernetes API server

The CCM consolidates all of the cloud-dependent logic from the preceding three components to create a single point of integration with the cloud. The new architecture with the CCM looks like this:



Components of the CCM

The CCM breaks away some of the functionality of Kubernetes controller manager (KCM) and runs it as a separate process. Specifically, it breaks away those controllers in the KCM that are cloud dependent. The KCM has the following cloud dependent controller loops:

- Node controller
- Volume controller
- Route controller
- Service controller

In version 1.9, the CCM runs the following controllers from the preceding list:

- Node controller
- Route controller
- Service controller

Additionally, it runs another controller called the PersistentVolumeLabels controller. This controller is responsible for setting the zone and region labels on PersistentVolumes created in GCP and AWS clouds.

> **Note:** Volume controller was deliberately chosen to not be a part of CCM. Due to the complexity involved and due to the existing efforts to abstract away vendor specific volume logic, it was decided that volume controller will not be moved to CCM.

The original plan to support volumes using CCM was to use Flex volumes to support pluggable volumes. However, a competing effort known as CSI is being planned to replace Flex.

Considering these dynamics, we decided to have an intermediate stop gap measure until CSI becomes ready.

Functions of the CCM

The CCM inherits its functions from components of Kubernetes that are dependent on a cloud provider. This section is structured based on those components.

1. Kubernetes controller manager

The majority of the CCM's functions are derived from the KCM. As mentioned in the previous section, the CCM runs the following control loops:

- Node controller
- Route controller
- Service controller
- PersistentVolumeLabels controller

Node controller

The Node controller is responsible for initializing a node by obtaining information about the nodes running in the cluster from the cloud provider. The node controller performs the following functions:

1. Initialize a node with cloud specific zone/region labels.
2. Initialize a node with cloud specific instance details, for example, type and size.
3. Obtain the node's network addresses and hostname.
4. In case a node becomes unresponsive, check the cloud to see if the node has been deleted from the cloud. If the node has been deleted from the cloud, delete the Kubernetes Node object.

Route controller

The Route controller is responsible for configuring routes in the cloud appropriately so that containers on different nodes in the Kubernetes cluster can communicate with each other. The route controller is only applicable for Google Compute Engine clusters.

Service Controller

The Service controller is responsible for listening to service create, update, and delete events. Based on the current state of the services in Kubernetes, it configures cloud load balancers (such as ELB or Google LB) to reflect the state of the services in Kubernetes. Additionally, it ensures that service backends for cloud load balancers are up to date.

PersistentVolumeLabels controller

The PersistentVolumeLabels controller applies labels on AWS EBS/GCE PD volumes when they are created. This removes the need for users to manually set the labels on these volumes.

These labels are essential for the scheduling of pods as these volumes are constrained to work only within the region/zone that they are in. Any Pod using these volumes needs to be scheduled in the same region/zone.

The PersistentVolumeLabels controller was created specifically for the CCM; that is, it did not exist before the CCM was created. This was done to move the PV labelling logic in the Kubernetes API server (it was an admission controller) to the CCM. It does not run on the KCM.

2. Kubelet

The Node controller contains the cloud-dependent functionality of the kubelet. Prior to the introduction of the CCM, the kubelet was responsible for initializing a node with cloud-specific details such as IP addresses, region/zone labels and instance type information. The introduction of the CCM has moved this initialization operation from the kubelet into the CCM.

In this new model, the kubelet initializes a node without cloud-specific information. However, it adds a taint to the newly created node that makes the node unschedulable until the CCM initializes the node with cloud-specific information. It then removes this taint.

3. Kubernetes API server

The PersistentVolumeLabels controller moves the cloud-dependent functionality of the Kubernetes API server to the CCM as described in the preceding sections.

Plugin mechanism

The cloud controller manager uses Go interfaces to allow implementations from any cloud to be plugged in. Specifically, it uses the CloudProvider Interface

defined here.

The implementation of the four shared controllers highlighted above, and some scaffolding along with the shared cloudprovider interface, will stay in the Kubernetes core. Implementations specific to cloud providers will be built outside of the core and implement interfaces defined in the core.

For more information about developing plugins, see Developing Cloud Controller Manager.

Authorization

This section breaks down the access required on various API objects by the CCM to perform its operations.

Node Controller

The Node controller only works with Node objects. It requires full access to get, list, create, update, patch, watch, and delete 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 for the Services, it requires access to create, list, get, watch, and update.

v1/Service:

- List
- Get
- Watch
- Patch
- Update

PersistentVolumeLabels controller

The PersistentVolumeLabels controller listens on PersistentVolume (PV) create events and then updates them. This controller requires access to get and update PVs.

v1/PersistentVolume:

- Get
- List
- Watch
- Update

Others

The implementation of the core of CCM requires access to create events, and to ensure secure operation, it requires access to create ServiceAccounts.

v1/Event:

- Create
- Patch
- Update

v1/ServiceAccount:

- Create

The RBAC ClusterRole for the CCM looks like this:

```
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
```

Vendor Implementations

The following cloud providers have implemented CCMs:

- Digital Ocean
- Oracle
- Azure
- GCE
- AWS

Cluster Administration

Complete instructions for configuring and running the CCM are provided here.

Edit This Page

Nodes

A node is a worker machine in Kubernetes, previously known as a `minion`. A node may be a VM or physical machine, depending on the cluster. Each node contains the services necessary to run pods and is managed by the master components. The services on a node include the container runtime, kubelet and kube-proxy. See The Kubernetes Node section in the architecture design doc for more details.

- Node Status
- Management
- API Object

Node Status

A node's status contains the following information:

- Addresses
- Condition
- Capacity
- Info

Each section is described in detail below.

Addresses

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.

Condition

The `conditions` field describes the status of all `Running` nodes.

Node Condition	Description
`OutOfDisk`	`True` if there is insufficient free space on the node for adding new pods, otherwise `Fal`
`Ready`	`True` if the node is healthy and ready to accept pods, `False` if the node is not healthy
`MemoryPressure`	`True` if pressure exists on the node memory – that is, if the node memory is low; othe
`PIDPressure`	`True` if pressure exists on the processes – that is, if there are too many processes on t
`DiskPressure`	`True` if pressure exists on the disk size – that is, if the disk capacity is low; otherwise
`NetworkUnavailable`	`True` if the network for the node is not correctly configured, otherwise `False`

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

```
"conditions": [
  {
    "type": "Ready",
    "status": "True"
  }
]
```

If the Status of the Ready condition remains `Unknown` or `False` for longer than the `pod-eviction-timeout`, an argument is passed to the kube-controller-manager and 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 apiserver 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 apiserver is re-established. In the meantime, the pods that are scheduled for deletion may continue to run

on the partitioned node.

In versions of Kubernetes prior to 1.5, the node controller would force delete these unreachable pods from the apiserver. However, in 1.5 and higher, 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 apiserver, and frees up their names.

In version 1.12, `TaintNodesByCondition` feature is promoted to beta so node lifecycle controller automatically creates taints that represent conditions. Similarly the scheduler ignores conditions when considering a Node; instead it looks at the Node's taints and a Pod's tolerations.

Now users can choose between the old scheduling model and a new, more flexible scheduling model. A Pod that does not have any tolerations gets scheduled according to the old model. But a Pod that tolerates the taints of a particular Node can be scheduled on that Node.

> **Caution:** Enabling this feature creates a small delay between the time when a condition is observed and when a taint is created. This delay is usually less than one second, but it can increase the number of Pods that are successfully scheduled but rejected by the kubelet.

Capacity

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

Info

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

Management

Unlike pods and services, a node is not inherently created by Kubernetes: it is created externally by cloud providers like Google Compute Engine, or it exists in your pool of physical or virtual machines. So when Kubernetes creates a node, it creates an object that represents the node. After creation, Kubernetes

checks whether the node is valid or not. For example, if you try to create a node from the following content:

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

Kubernetes creates a node object internally (the representation), and validates the node by health checking based on the `metadata.name` field. If the node is valid – that is, if all necessary services are running – it is eligible to run a pod. Otherwise, it is ignored for any cluster activity until it becomes valid.

> **Note:** Kubernetes keeps the object for the invalid node and keeps checking to see whether it becomes valid. You must explicitly delete the Node object to stop this process.

Currently, there are three components that interact with the Kubernetes node interface: node controller, kubelet, and kubectl.

Node Controller

The node controller is a Kubernetes master component which 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, e.g. 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.

In Kubernetes 1.4, we updated the logic of the node controller to better handle cases when a large number of nodes have problems with reaching the master (e.g. because the master has networking problem). Starting with 1.4, the node controller looks at the state of all nodes in the cluster when making a decision about pod eviction.

In most cases, 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 node controller evicts at the normal rate `--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 case, the node controller assumes that there's some problem with master connectivity and stops all evictions until some connectivity is restored.

Starting in Kubernetes 1.6, the NodeController is also responsible for evicting pods that are running on nodes with `NoExecute` taints, when the pods do not tolerate the taints. Additionally, as an alpha feature that is disabled by default, the NodeController is responsible for adding taints corresponding to node problems like node unreachable or not ready. See this documentation for details about `NoExecute` taints and the alpha feature.

Starting in version 1.8, the node controller can be made responsible for creating taints that represent Node conditions. This is an alpha feature of version 1.8.

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 apiserver.
- `--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.
- `--node-status-update-frequency` - Specifies how often kubelet posts node status to master.

Currently, any kubelet is authorized to create/modify any node resource, but in practice it only creates/modifies its own. (In the future, we plan to only allow a kubelet to modify its own node resource.)

Manual Node Administration

A cluster administrator can create and modify node objects.

If the administrator wishes to create node objects manually, set the kubelet flag `--register-node=false`.

The administrator can modify node resources (regardless of the setting of `--register-node`). Modifications include setting labels on the node and marking it unschedulable.

Labels on nodes can be used in conjunction with node selectors on pods to control scheduling, e.g. to constrain a pod to only be eligible to run on a subset of the nodes.

Marking a node as unschedulable prevents new pods from being scheduled to that node, but does not affect any existing pods on the node. This is useful as a preparatory step before a node reboot, etc. For example, to mark a node unschedulable, run this command:

```
kubectl cordon $NODENAME
```

> **Note:** Pods created by a DaemonSet controller bypass the Kubernetes scheduler and do not respect the unschedulable attribute on a node. This assumes that daemons belong on the machine even if it is being drained of applications while it prepares for a reboot.

Node capacity

The capacity of the node (number of cpus and amount of memory) is part of the node object. Normally, nodes register themselves and report their capacity when creating the node object. If you are doing manual node administration, then you need to set node capacity when adding a node.

The Kubernetes scheduler ensures that there are enough resources for all the pods on a node. It checks that the sum of the requests of containers on the node is no greater than the node capacity. It includes all containers started by the kubelet, but not containers started directly by the container runtime nor any process running outside of the containers.

If you want to explicitly reserve resources for non-pod processes, you can create a placeholder pod. Use the following template:

```
apiVersion: v1
kind: Pod
metadata:
  name: resource-reserver
spec:
  containers:
  - name: sleep-forever
    image: k8s.gcr.io/pause:0.8.0
    resources:
      requests:
        cpu: 100m
        memory: 100Mi
```

Set the `cpu` and `memory` values to the amount of resources you want to reserve. Place the file in the manifest directory (`--config=DIR` flag of kubelet). Do this on each kubelet where you want to reserve resources.

API Object

Node is a top-level resource in the Kubernetes REST API. More details about the API object can be found at: Node API object.

Edit This Page

Master-Node communication

This document catalogs the communication paths between the master (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).

- Cluster to Master
- Master to Cluster

Cluster to Master

All communication paths from the cluster to the master terminate at the apiserver (none of the other master components are designed to expose remote services). In a typical deployment, the apiserver is configured to listen for remote connections on a secure HTTPS port (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. For example, on a default GKE deployment, 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 master components also communicate with the cluster apiserver over the secure port.

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

Master to Cluster

There are two primary communication paths from the master (apiserver) to the cluster. 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.

Edit This Page

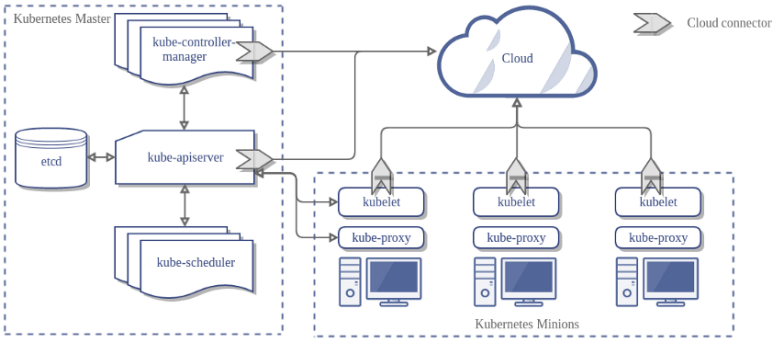
Concepts Underlying the Cloud Controller Manager

The cloud controller manager (CCM) concept (not to be confused with the binary) was originally created to allow cloud specific vendor code and the Kubernetes core to evolve independent of one another. The cloud controller manager runs alongside other master components such as the Kubernetes controller manager, the API server, and scheduler. It can also be started as a Kubernetes addon, in which case it runs on top of Kubernetes.

The cloud controller manager's design is based on a plugin mechanism that allows new cloud providers to integrate with Kubernetes easily by using plugins. There are plans in place for on-boarding new cloud providers on Kubernetes and for migrating cloud providers from the old model to the new CCM model.

This document discusses the concepts behind the cloud controller manager and gives details about its associated functions.

Here's the architecture of a Kubernetes cluster without the cloud controller manager:



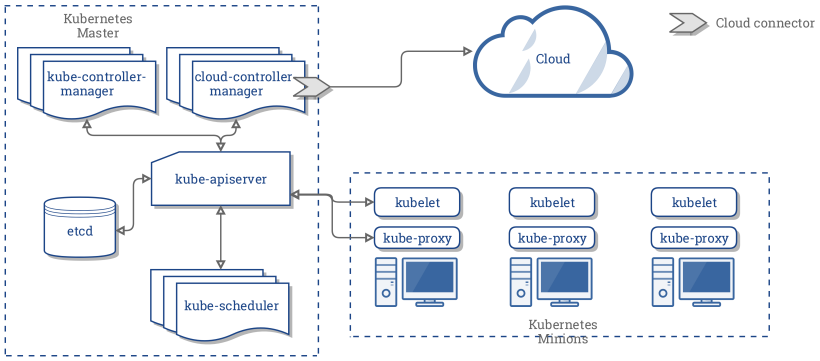
- Design
- Components of the CCM
- Functions of the CCM
- Plugin mechanism
- Authorization
- Vendor Implementations
- Cluster Administration

Design

In the preceding diagram, Kubernetes and the cloud provider are integrated through several different components:

- Kubelet
- Kubernetes controller manager
- Kubernetes API server

The CCM consolidates all of the cloud-dependent logic from the preceding three components to create a single point of integration with the cloud. The new architecture with the CCM looks like this:



Components of the CCM

The CCM breaks away some of the functionality of Kubernetes controller manager (KCM) and runs it as a separate process. Specifically, it breaks away those controllers in the KCM that are cloud dependent. The KCM has the following cloud dependent controller loops:

- Node controller
- Volume controller
- Route controller
- Service controller

In version 1.9, the CCM runs the following controllers from the preceding list:

- Node controller
- Route controller
- Service controller

Additionally, it runs another controller called the PersistentVolumeLabels controller. This controller is responsible for setting the zone and region labels on PersistentVolumes created in GCP and AWS clouds.

> **Note:** Volume controller was deliberately chosen to not be a part of CCM. Due to the complexity involved and due to the existing efforts to abstract away vendor specific volume logic, it was decided that volume controller will not be moved to CCM.

The original plan to support volumes using CCM was to use Flex volumes to support pluggable volumes. However, a competing effort known as CSI is being planned to replace Flex.

Considering these dynamics, we decided to have an intermediate stop gap measure until CSI becomes ready.

Functions of the CCM

The CCM inherits its functions from components of Kubernetes that are dependent on a cloud provider. This section is structured based on those components.

1. Kubernetes controller manager

The majority of the CCM's functions are derived from the KCM. As mentioned in the previous section, the CCM runs the following control loops:

- Node controller
- Route controller
- Service controller
- PersistentVolumeLabels controller

Node controller

The Node controller is responsible for initializing a node by obtaining information about the nodes running in the cluster from the cloud provider. The node controller performs the following functions:

1. Initialize a node with cloud specific zone/region labels.
2. Initialize a node with cloud specific instance details, for example, type and size.
3. Obtain the node's network addresses and hostname.
4. In case a node becomes unresponsive, check the cloud to see if the node has been deleted from the cloud. If the node has been deleted from the cloud, delete the Kubernetes Node object.

Route controller

The Route controller is responsible for configuring routes in the cloud appropriately so that containers on different nodes in the Kubernetes cluster can communicate with each other. The route controller is only applicable for Google Compute Engine clusters.

Service Controller

The Service controller is responsible for listening to service create, update, and delete events. Based on the current state of the services in Kubernetes, it configures cloud load balancers (such as ELB or Google LB) to reflect the state of the services in Kubernetes. Additionally, it ensures that service backends for cloud load balancers are up to date.

PersistentVolumeLabels controller

The PersistentVolumeLabels controller applies labels on AWS EBS/GCE PD volumes when they are created. This removes the need for users to manually set the labels on these volumes.

These labels are essential for the scheduling of pods as these volumes are constrained to work only within the region/zone that they are in. Any Pod using these volumes needs to be scheduled in the same region/zone.

The PersistentVolumeLabels controller was created specifically for the CCM; that is, it did not exist before the CCM was created. This was done to move the PV labelling logic in the Kubernetes API server (it was an admission controller) to the CCM. It does not run on the KCM.

2. Kubelet

The Node controller contains the cloud-dependent functionality of the kubelet. Prior to the introduction of the CCM, the kubelet was responsible for initializing a node with cloud-specific details such as IP addresses, region/zone labels and instance type information. The introduction of the CCM has moved this initialization operation from the kubelet into the CCM.

In this new model, the kubelet initializes a node without cloud-specific information. However, it adds a taint to the newly created node that makes the node unschedulable until the CCM initializes the node with cloud-specific information. It then removes this taint.

3. Kubernetes API server

The PersistentVolumeLabels controller moves the cloud-dependent functionality of the Kubernetes API server to the CCM as described in the preceding sections.

Plugin mechanism

The cloud controller manager uses Go interfaces to allow implementations from any cloud to be plugged in. Specifically, it uses the CloudProvider Interface defined here.

The implementation of the four shared controllers highlighted above, and some scaffolding along with the shared cloudprovider interface, will stay in the Kubernetes core. Implementations specific to cloud providers will be built outside of the core and implement interfaces defined in the core.

For more information about developing plugins, see Developing Cloud Controller Manager.

Authorization

This section breaks down the access required on various API objects by the CCM to perform its operations.

Node Controller

The Node controller only works with Node objects. It requires full access to get, list, create, update, patch, watch, and delete 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 for the Services, it requires access to create, list, get, watch, and update.

v1/Service:

- List
- Get
- Watch
- Patch
- Update

PersistentVolumeLabels controller

The PersistentVolumeLabels controller listens on PersistentVolume (PV) create events and then updates them. This controller requires access to get and update PVs.

v1/PersistentVolume:

- Get
- List
- Watch
- Update

Others

The implementation of the core of CCM requires access to create events, and to ensure secure operation, it requires access to create ServiceAccounts.

v1/Event:

- Create
- Patch
- Update

v1/ServiceAccount:

- Create

The RBAC ClusterRole for the CCM looks like this:

```
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
```

Vendor Implementations

The following cloud providers have implemented CCMs:

- Digital Ocean
- Oracle
- Azure
- GCE
- AWS

Cluster Administration

Complete instructions for configuring and running the CCM are provided here.

Edit This Page

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 OperatorsA person who configures, controls, and monitors clusters. who want to understand how to adapt their Kubernetes cluster to the needs of their work environment. Developers who are prospective Platform DevelopersA person who customizes the Kubernetes platform to fit the needs of their project. or Kubernetes Project ContributorsSomeone who donates code, documentation, or their time to help the Kubernetes project or community. will also find it useful as an introduction to what extension points and patterns exist, and their trade-offs and limitations.

- Overview
- Configuration
- Extensions
- Extension Patterns
- Extension Points
- API Extensions
- Infrastructure Extensions
- What's next

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 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.

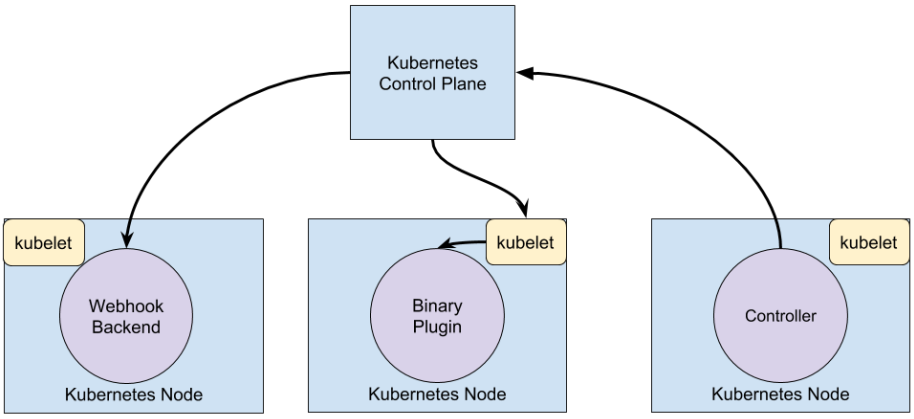
There is a specific pattern for writing client programs that work well with Kubernetes called the *Controller* pattern. Controllers typically read an object's

`.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.

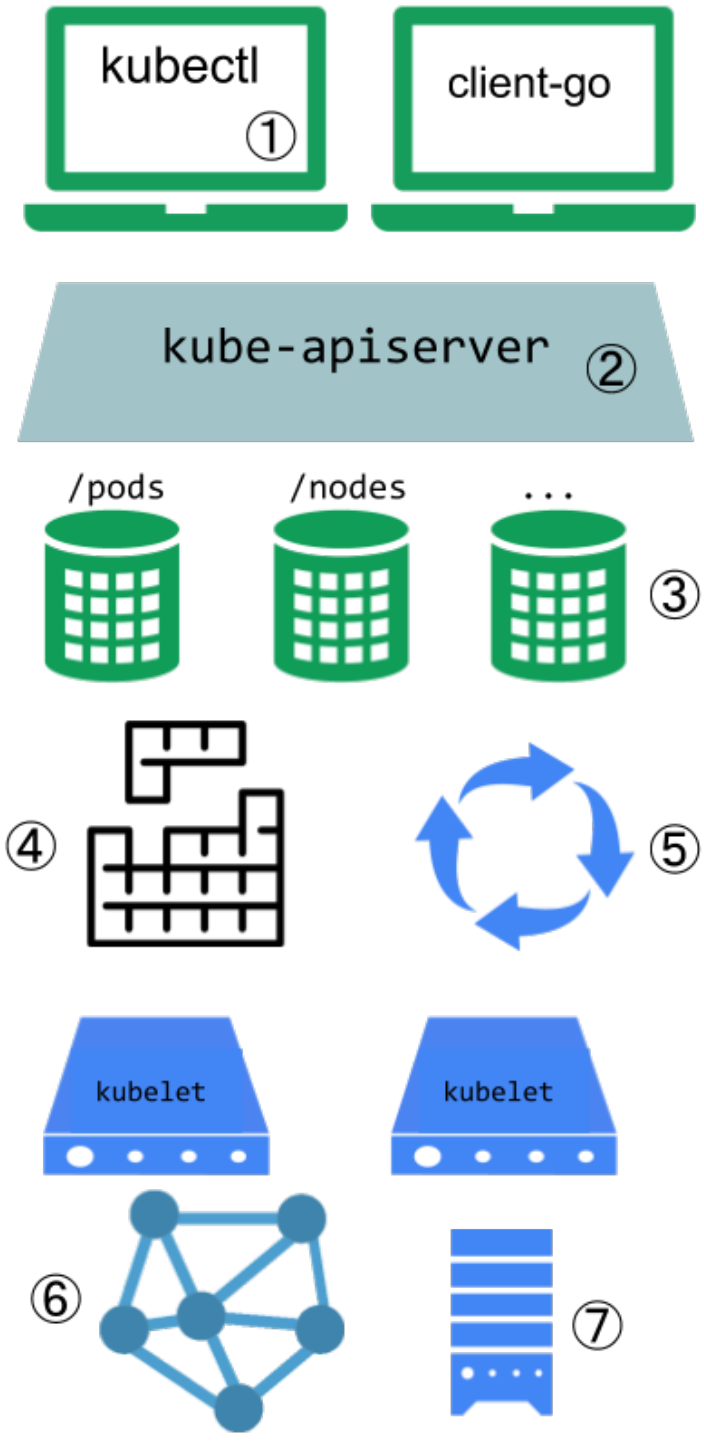
In the webhook model, Kubernetes makes a network request to a remote service. In the *Binary Plugin* model, Kubernetes executes a binary (program). Binary plugins are used by the kubelet (e.g. Flex Volume Plugins and Network Plugins) and by kubectl.

Below is a diagram showing how the extensions points interact with the Kubernetes control plane.



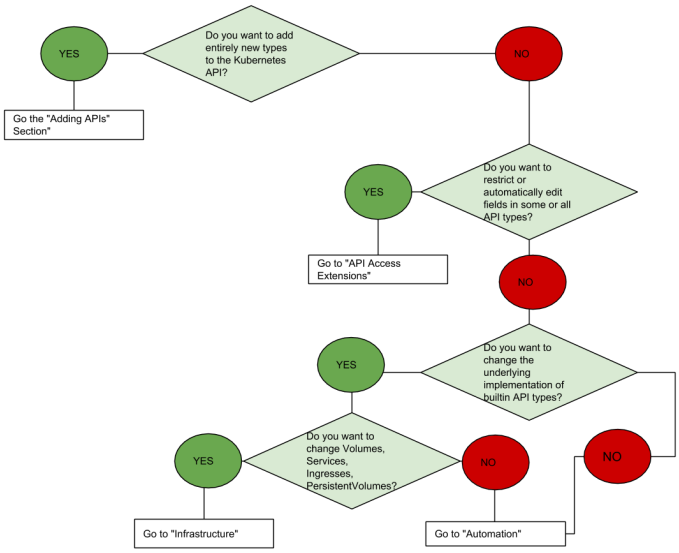
Extension Points

This diagram shows the extension points in a Kubernetes system.


kubectl
①
client-go
kube-apiserver
②
/pods
/nodes
...
③
④
⑤
kubelet
kubelet
⑥
⑦

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

Often, when you add a new API, you also add a control loop that reads and/or writes the new APIs. When the combination of a Custom API and a control loop is used to manage a specific, usually stateful, application, this is called the *Operator* pattern. Custom APIs and control loops can also be used to control other resources, such as storage, policies, and so on.

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.

- Initializers are controllers that can modify objects before they are created. Initializers can modify initial object creations but cannot affect updates to objects. Initializers can also reject objects.

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
- See examples of Automation
  - List of Operators

Edit This Page

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 OperatorsA person who configures, controls, and monitors clusters. who want to understand how to adapt their Kubernetes cluster to the needs of their work environment. Developers who are prospective Platform DevelopersA person who customizes the Kubernetes platform to fit the needs of their project. or Kubernetes Project ContributorsSomeone who donates code, documentation, or their time to help the Kubernetes project or community. will also find it useful as an introduction to what extension points and patterns exist, and their trade-offs and limitations.

- Overview
- Configuration
- Extensions
- Extension Patterns
- Extension Points
- API Extensions
- Infrastructure Extensions
- What's next

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 need to install extensions and fewer will need to author new ones.

Extension Patterns

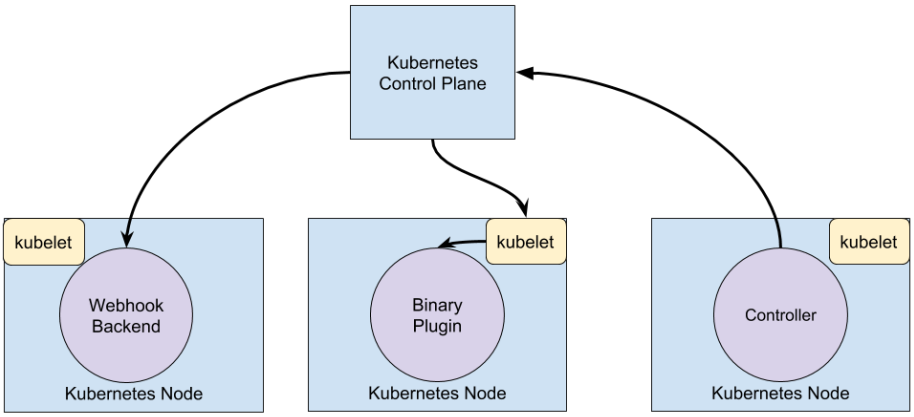
Kubernetes is designed to be automated by writing client programs. Any program that reads and/or writes to the Kubernetes API can provide useful automation. *Automation* can run on the cluster or off it. By following the guidance in this doc you can write highly available and robust automation. Automation 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.

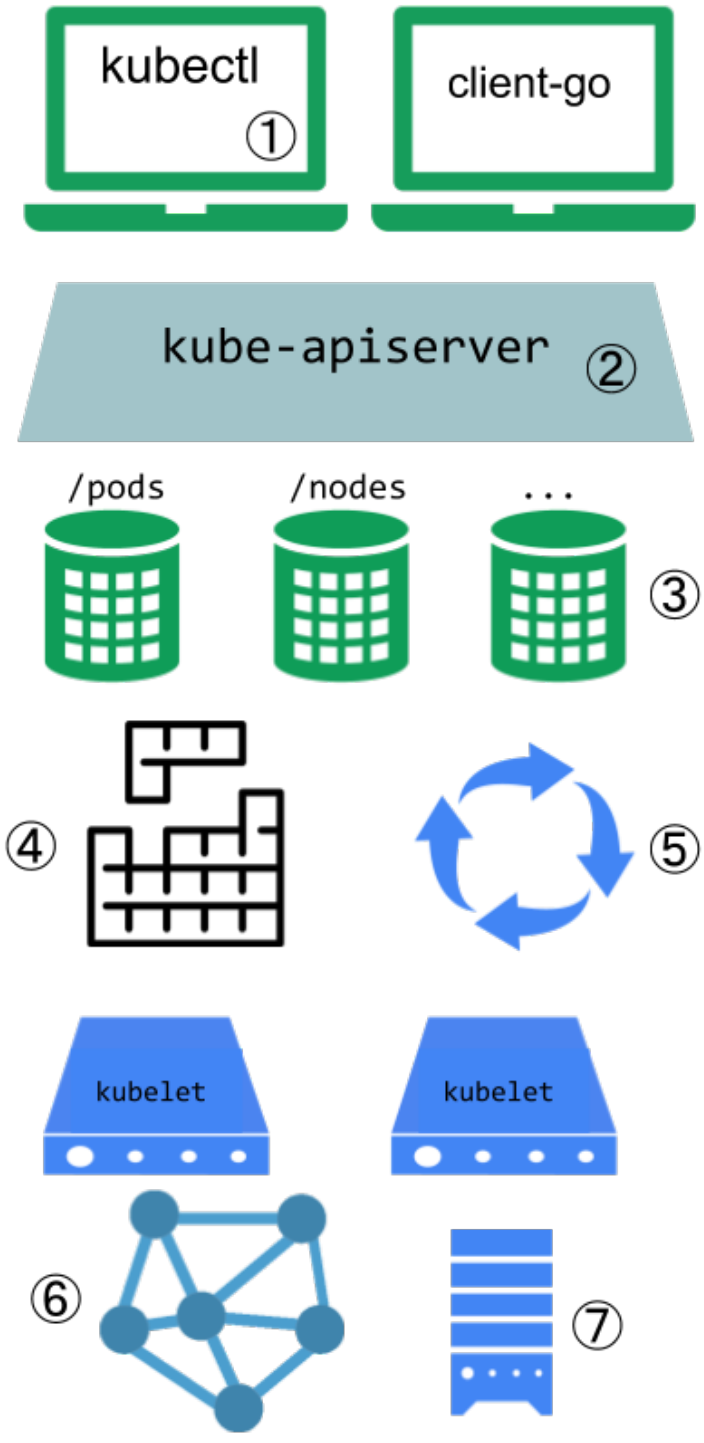
In the webhook model, Kubernetes makes a network request to a remote service. In the *Binary Plugin* model, Kubernetes executes a binary (program). Binary plugins are used by the kubelet (e.g. Flex Volume Plugins and Network Plugins) and by kubectl.

Below is a diagram showing how the extensions points interact with the Kubernetes control plane.



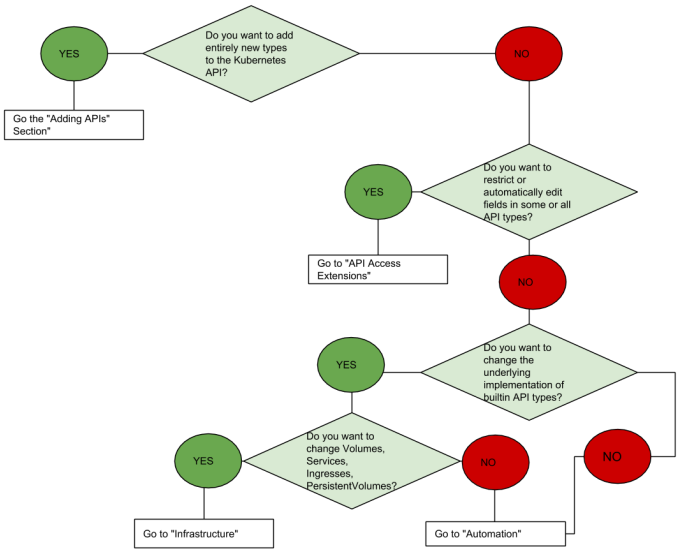
Extension Points

This diagram shows the extension points in a Kubernetes system.


kubectl
①
client-go
kube-apiserver
②
/pods
/nodes
...
③
④
⑤
kubelet
kubelet
⑥
⑦

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

Often, when you add a new API, you also add a control loop that reads and/or writes the new APIs. When the combination of a Custom API and a control loop is used to manage a specific, usually stateful, application, this is called the *Operator* pattern. Custom APIs and control loops can also be used to control other resources, such as storage, policies, and so on.

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.

- Initializers are controllers that can modify objects before they are created. Initializers can modify initial object creations but cannot affect updates to objects. Initializers can also reject objects.

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
- See examples of Automation
  - List of Operators

Edit This Page

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.

- Overview
- What's next

Overview

The aggregation layer enables installing additional Kubernetes-style APIs in your cluster. These can either be pre-built, existing 3rd party solutions, such as service-catalog, or user-created APIs like apiserver-builder, which can get you started.

In 1.7 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, users must 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.

Ordinarily, the APIService will be implemented by an *extension-apiserver* in a pod running in the cluster. This extension-apiserver will normally need to be paired with one or more controllers if active management of the added resources is needed. As a result, the apiserver-builder will actually provide a skeleton for both. As another example, when the service-catalog is installed, it provides both the extension-apiserver and controller for the services it provides.

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.

Edit This Page

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.

- Overview
- What's next

Overview

The aggregation layer enables installing additional Kubernetes-style APIs in your cluster. These can either be pre-built, existing 3rd party solutions, such as service-catalog, or user-created APIs like apiserver-builder, which can get you started.

In 1.7 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, users must 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.

Ordinarily, the APIService will be implemented by an *extension-apiserver* in a pod running in the cluster. This extension-apiserver will normally need to be paired with one or more controllers if active management of the added resources is needed. As a result, the apiserver-builder will actually provide a skeleton for both. As another example, when the service-catalog is installed, it provides both the extension-apiserver and controller for the services it provides.

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.

Edit This Page

Custom Resources

This page explains *custom resources*, which are extensions of the Kubernetes API, including 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
- Adding custom resources
- CustomResourceDefinitions
- API server aggregation
- Preparing to install a custom resource
- Accessing a custom resource
- What's next

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 on every Kubernetes cluster. In other words, it represents a customization of a particular Kubernetes installation.

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 with 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. It is only when combined with a *controller* that they become a true declarative API. A declarative API allows you to *declare* or specify the desired state of your resource and tries to match the actual state to this desired state. Here, the controller interprets the structured data as a record of the user's desired state, and continually takes action to achieve and maintain this state.

A *custom controller* is a controller that users can deploy and update on a running cluster, independently of the cluster's own lifecycle. Custom controllers can work with any kind of resource, but they are especially effective when combined with custom resources. The Operator pattern is one example of such a combination. It allows developers 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:
Your API is Declarative.
You want your new types to be readable and writable using `kubectl`.
You want to view your new types in a Kubernetes UI, such as dashboard, alongside built-in types.
You are developing a new API.
You are willing to accept the format restriction that Kubernetes puts on REST resource paths, such as API G
Your resources are naturally scoped to a cluster or to namespaces of a cluster.
You want to reuse Kubernetes API support 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 objects to determine completion of the request.
- You talk about Remote Procedure Calls (RPCs).
- Directly storing large amounts of data (e.g. > 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 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 APIServers 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 APIserver. 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.

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, which uses Custom Resources for a demonstration of how to register a new custom resource, work with instances of your new resource type, and setup a controller to handle events.

> **Note:** CRD is the successor to the deprecated *ThirdPartyResource* (TPR) API, and is available as of Kubernetes 1.7.

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 handle custom resources in a generic way 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
Do not require programming. Users can choose any language for a CRD controller.
No additional service to run; CRs are handled by API Server.
No ongoing support once the CRD is created. Any bug fixes are picked up as part of normal Kubernetes Mas
No need to handle multiple versions of your API. For example: when you control the client for this resource,

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	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.	Yes. Most validation can be specified in the CRD using OpenAPI v3.0 validation. Any other validations supported by addition of a Validating Webhook.	Yes, arbitrary validation checks
Defaulting	See above	Yes, via a Mutating Webhook; Planned, via CRD OpenAPI schema.	Yes
Multi-versioning	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.	Yes	Yes

Feature	Description	CRDs	Aggregated API
Custom Storage	If you need storage with a different performance mode (for example, time-series database instead of key-value store) or isolation for security (for example, encryption secrets or different	No	Yes
Custom Business Logic	Perform arbitrary checks or actions when creating, reading, updating or deleting an object	Yes, using Webhooks.	Yes
Scale Subresource	Allows systems like HorizontalPodAutoscaler and PodDisruptionBudget interact with your new resource	Yes	Yes

Feature	Description	CRDs	Aggregated API
Status Subresource	• Finer-grained access control: user writes spec section, controller writes status section. • Allows incrementing object Generation on custom resource data mutation (requires separate spec and status sections in the resource)	Yes	Yes
Other Subresources	Add operations other than CRUD, such as "logs" or "exec".	No	Yes

Feature	Description	CRDs	Aggregated API
strategic-merge-patch	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"	No	Yes
Protocol Buffers	The new resource supports clients that want to use Protocol Buffers	No	Yes
OpenAPI Schema	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?)	No, but planned	Yes

Common Features

When you create a custom resource, either via a CRDs or an AA, you get many features for your API, compared to implementing it outside the Kubernetes platform:

Feature	What it does
CRUD	The new endpoints support CRUD basic operations via HTTP and `kubectl`
Watch	The new endpoints support Kubernetes Watch operations via HTTP
Discovery	Clients like kubectl and dashboard automatically offer list, display, and field ed
json-patch	The new endpoints support PATCH with `Content-Type: application/json-`
merge-patch	The new endpoints support PATCH with `Content-Type: application/merge`
HTTPS	The new endpoints uses HTTPS
Built-in Authentication	Access to the extension uses the core apiserver (aggregation layer) for authentic
Built-in Authorization	Access to the extension can reuse the authorization used by the core apiserver (
Finalizers	Block deletion of extension resources until external cleanup happens.
Admission Webhooks	Set default values and validate extension resources during any create/update/d
UI/CLI Display	Kubectl, dashboard can display extension resources.
Unset vs Empty	Clients can distinguish unset fields from zero-valued fields.
Client Libraries Generation	Kubernetes provides generic client libraries, as well as tools to generate type-sp
Labels and annotations	Common metadata across objects that tools know how to edit for core and cust

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 APIserver 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.

Edit This Page

Device Plugins

Starting in version 1.8, Kubernetes provides a device plugin framework for vendors to advertise their resources to the kubelet without changing Kubernetes core code. Instead of writing custom Kubernetes code, vendors can implement a device plugin that can be deployed manually or as a DaemonSet. The targeted devices include GPUs, High-performance NICs, FPGAs, InfiniBand, and other

similar computing resources that may require vendor specific initialization and setup.

- Device plugin registration
- Device plugin implementation
- Device plugin deployment
- Examples

Device plugin registration

The device plugins feature is gated by the `DevicePlugins` feature gate which is disabled by default before 1.10. When the device plugins feature is enabled, 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`. 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 `vendor-domain/foo` with the kubelet and reports two healthy devices on a node, the node status is updated to advertise 2 `vendor-domain/foo`.

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 `vendor-domain/resource` on certain nodes, here is an example user pod requesting this resource:

```
apiVersion: v1
kind: Pod
metadata:
  name: demo-pod
```

```
spec:
  containers:
    - name: demo-container-1
      image: k8s.gcr.io/pause:2.0
      resources:
        limits:
          vendor-domain/resource: 2 # requesting 2 vendor-domain/resource
```

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 {
      // 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) {}
}
```

- 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.

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

A device plugin can be deployed manually or as a DaemonSet. Being deployed as a DaemonSet has the benefit that Kubernetes can restart the device plugin if it fails. Otherwise, an extra mechanism is needed to recover from device plugin failures. The canonical directory `/var/lib/kubelet/device-plugins` requires privileged access, so a device plugin must run in a privileged security context. If a device plugin is running as a DaemonSet, `/var/lib/kubelet/device-plugins` must be mounted as a Volume in the plugin's PodSpec.

Kubernetes device plugin support is still in alpha. As development continues, its API version can change in incompatible ways. We recommend that device plugin developers do the following:

- 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 to ensure the continuous functioning of the device allocations during the upgrade.

Examples

For examples of device plugin implementations, see:

- The official NVIDIA GPU device plugin
  - Requires nvidia-docker 2.0 which allows you to run GPU enabled docker containers.
  - A detailed guide on how to schedule NVIDIA GPUs on k8s.
- The NVIDIA GPU device plugin for COS base OS
- The RDMA device plugin
- The Solarflare device plugin
- The AMD GPU device plugin
- The SRIOV Network device plugin
- The Intel device plugins for GPU, FPGA and QuickAssist devices

FEATURE STATE: `Kubernetes v1.12` beta

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

- The version names contain beta (e.g. v2beta3).
- Code is well tested. Enabling the feature is considered safe. Enabled by default.
- Support for the overall feature will not be dropped, though details may change.
- The schema and/or semantics of objects may change in incompatible ways in a subsequent beta or stable release. When this happens, we will provide instructions for migrating to the next version. This may require deleting, editing, and re-creating API objects. The editing process may require some thought. This may require downtime for applications that rely on the feature.
- Recommended for only non-business-critical uses because of potential for incompatible changes in subsequent releases. If you have multiple clusters that can be upgraded independently, you may be able to relax this restriction.
- **Please do try our beta features and give feedback on them! After they exit beta, it may not be practical for us to make more changes.**

Edit This Page

Network Plugins

FEATURE STATE: `Kubernetes v1.12` alpha

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

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

Warning: Alpha features change rapidly.

Network plugins in Kubernetes come in a few flavors:

- CNI plugins: adhere to the appc/CNI specification, designed for interoperability.
- Kubenet plugin: implements basic `cbr0` using the `bridge` and `host-local` CNI plugins

- Installation
- Network Plugin Requirements

- Usage Summary

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 found, and executes the selected plugin at appropriate times in the pod lifecycle (this is only true for Docker, as rkt 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 first one in lexicographic order of file name is used.

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

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 a `bandwidth` plugin to your CNI configuration file (default `/etc/cni/net.d`).

```
{
  "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 e.g. `--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.

Edit This Page

Device Plugins

Starting in version 1.8, Kubernetes provides a device plugin framework for vendors to advertise their resources to the kubelet without changing Kubernetes core code. Instead of writing custom Kubernetes code, vendors can implement a device plugin that can be deployed manually or as a DaemonSet. The targeted devices include GPUs, High-performance NICs, FPGAs, InfiniBand, and other similar computing resources that may require vendor specific initialization and setup.

- Device plugin registration
- Device plugin implementation
- Device plugin deployment
- Examples

Device plugin registration

The device plugins feature is gated by the `DevicePlugins` feature gate which is disabled by default before 1.10. When the device plugins feature is enabled, 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`. 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 `vendor-domain/foo` with the kubelet and reports two healthy devices on a node, the node status is updated to advertise 2 `vendor-domain/foo`.

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 `vendor-domain/resource` on certain nodes, here is an example user pod requesting this resource:

```
apiVersion: v1
kind: Pod
metadata:
  name: demo-pod
spec:
  containers:
    - name: demo-container-1
      image: k8s.gcr.io/pause:2.0
      resources:
        limits:
          vendor-domain/resource: 2 # requesting 2 vendor-domain/resource
```

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 {
      // 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) {}
}
```

- 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.

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

A device plugin can be deployed manually or as a DaemonSet. Being deployed as a DaemonSet has the benefit that Kubernetes can restart the device plugin if it fails. Otherwise, an extra mechanism is needed to recover from device plugin failures. The canonical directory `/var/lib/kubelet/device-plugins` requires privileged access, so a device plugin must run in a privileged security context. If a device plugin is running as a DaemonSet, `/var/lib/kubelet/device-plugins` must be mounted as a Volume in the plugin's PodSpec.

Kubernetes device plugin support is still in alpha. As development continues, its API version can change in incompatible ways. We recommend that device plugin developers do the following:

- 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 to ensure the continuous functioning of the device allocations during the upgrade.

Examples

For examples of device plugin implementations, see:

- The official NVIDIA GPU device plugin
  - Requires nvidia-docker 2.0 which allows you to run GPU enabled docker containers.

  - A detailed guide on how to schedule NVIDIA GPUs on k8s.
- The NVIDIA GPU device plugin for COS base OS
- The RDMA device plugin
- The Solarflare device plugin
- The AMD GPU device plugin
- The SRIOV Network device plugin
- The Intel device plugins for GPU, FPGA and QuickAssist devices

FEATURE STATE: `Kubernetes v1.12` beta

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

- The version names contain beta (e.g. v2beta3).
- Code is well tested. Enabling the feature is considered safe. Enabled by default.
- Support for the overall feature will not be dropped, though details may change.
- The schema and/or semantics of objects may change in incompatible ways in a subsequent beta or stable release. When this happens, we will provide instructions for migrating to the next version. This may require deleting, editing, and re-creating API objects. The editing process may require some thought. This may require downtime for applications that rely on the feature.
- Recommended for only non-business-critical uses because of potential for incompatible changes in subsequent releases. If you have multiple clusters that can be upgraded independently, you may be able to relax this restriction.
- **Please do try our beta features and give feedback on them! After they exit beta, it may not be practical for us to make more changes.**

Edit This Page

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 ServicesA software offering maintained by a third-party provider. from Service BrokersAn endpoint for a set of Managed Services offered and maintained by a third-party. without needing detailed knowledge about how those services are created or managed.

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 operatorA person who configures, controls, and monitors clusters. 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
- Architecture
- Usage
- What's next

Example use case

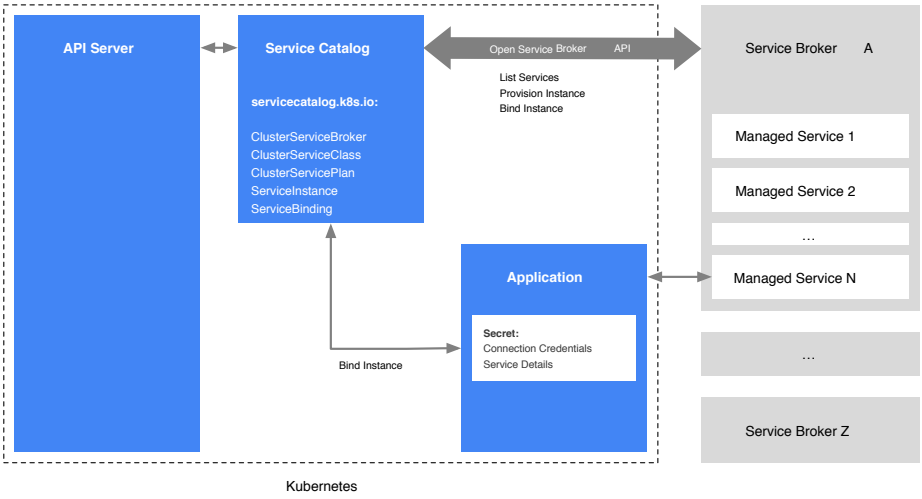
An application developerA person who writes an application that runs in a Kubernetes cluster. wants to use message queuing as part of their application running in a Kubernetes cluster. However, they do not want to deal with the overhead of setting such a service up and administering it themselves. Fortunately, 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.

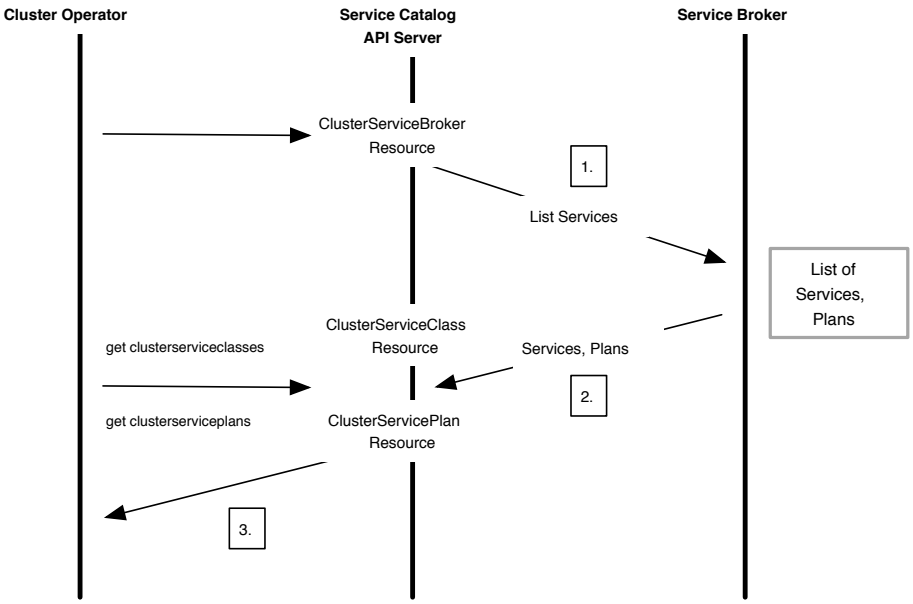
Listing managed services and Service Plans

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/broker
  #####
  # 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,EXTERN
   ```

   It should output a list of service names with a format similar to:

   ```
   SERVICE NAME                            EXTERNAL NAME
   4f6e6cf6-ffdd-425f-a2c7-3c9258ad2468    cloud-provider-service
   ...                                     ...
   ```

   They can also view the Service Plans available using the following command:

   ```
   kubectl get clusterserviceplans -o=custom-columns=PLAN\ NAME:.metadata.name,EXTERNAL\ N
   ```

   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
   ...                                     ...
   ```

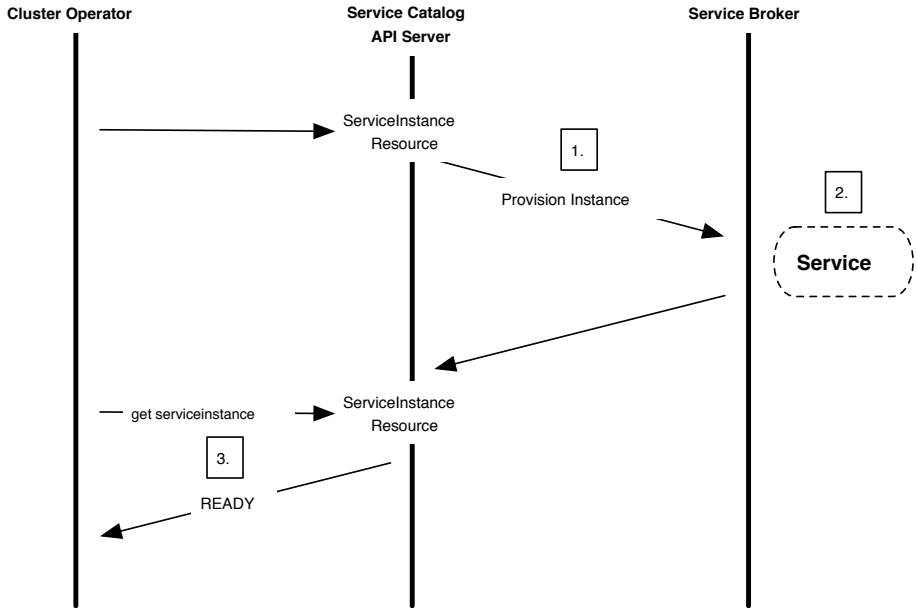
Provisioning a new instance

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.

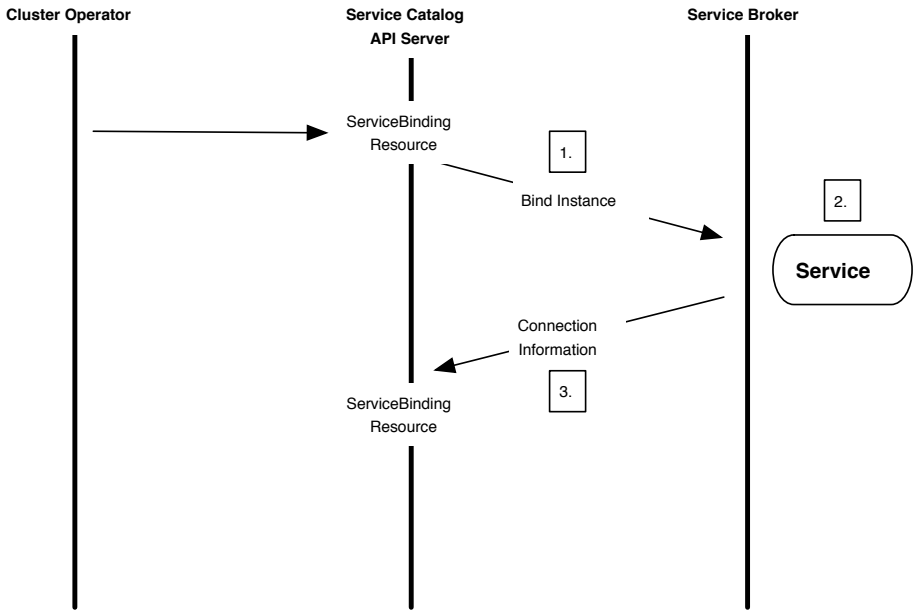
Binding to a managed service

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.
  #####
```

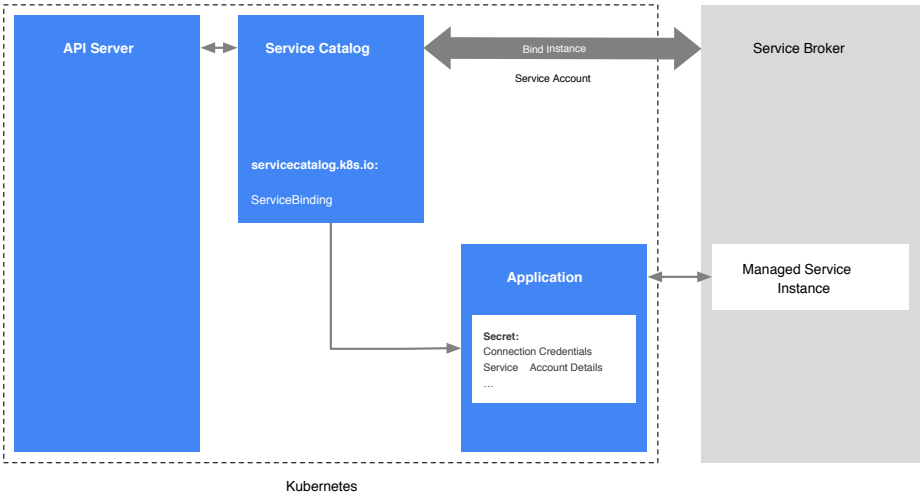
The following sequence diagram illustrates the steps involved in binding to a managed service instance:



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 ChartsA package of pre-configured Kubernetes resources that can be managed with the Helm tool., 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-incubator/service-catalog project.
- View svc-cat.io.

Edit This Page

Container Environment Variables

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

- Container environment
- What's next

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.

What's next

- Learn more about Container lifecycle hooks.
- Get hands-on experience attaching handlers to Container lifecycle events.

Edit This Page

Images

You create your Docker image and push it to a registry before referring to it in a Kubernetes pod.

The `image` property of a container supports the same syntax as the `docker` command does, including private registries and tags.

- Updating Images
- Building Multi-architecture Images with Manifests
- Using a Private Registry

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.

Note that you should avoid using `:latest` tag, see Best Practices for Configuration for more information.

Building Multi-architecture Images with Manifests

Docker CLI now supports the following command `docker manifest` with sub commands like `create`, `annotate` and `push`. These commands can be used to build and push the manifests. You can use `docker manifest inspect` to view the manifest.

Please see docker documentation here: https://docs.docker.com/edge/engine/reference/commandline/manifest/

See examples on how we use this in our build harness: https://cs.k8s.io/?q=docker%20manifest%20(create%7Cpush%7Cannotate)&i=nope&files=&repos=

These commands rely on and are implemented purely on the Docker CLI. You will need to either edit the `$HOME/.docker/config.json` and set `experimental` key to `enabled` or you can just set `DOCKER_CLI_EXPERIMENTAL` environment variable to `enabled` when you call the CLI commands.

> **Note:** Please use Docker *18.06 or above*, versions below that either have bugs or do not support the experimental command line option. Example https://github.com/docker/cli/issues/1135 causes problems under containerd.

If you run into trouble with uploading stale manifests, just clean up the older manifests in `$HOME/.docker/manifests` to start fresh.

For Kubernetes, we have typically used images with 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:

- Using Google Container Registry
  - Per-cluster
  - automatically configured on Google Compute Engine or Google Kubernetes Engine
  - all pods can read the project's private registry
- Using AWS EC2 Container Registry (ECR)
  - use IAM roles and policies to control access to ECR repositories
  - automatically refreshes ECR login credentials
- Using Azure Container Registry (ACR)
- Using IBM Cloud Container Registry
- Configuring Nodes to Authenticate to a Private Registry
  - all pods can read any configured private registries
  - requires node configuration by cluster administrator
- Pre-pulling 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

Each option is described in more detail below.

Using Google Container Registry

Kubernetes has native support for the Google Container Registry (GCR), when running on Google Compute Engine (GCE). If you are running your cluster on GCE or Google Kubernetes Engine, simply use the full image name (e.g. gcr.io/my_project/image:tag).

All pods in a cluster will have read access to images in this registry.

The kubelet will authenticate to GCR using the instance's Google service account. The service account on the instance will have a `https://www.googleapis.com/auth/devstorage.read_` so it can pull from the project's GCR, but not push.

Using AWS EC2 Container Registry

Kubernetes has native support for the AWS EC2 Container Registry, when nodes are AWS EC2 instances.

Simply use the full image name (e.g. `ACCOUNT.dkr.ecr.REGION.amazonaws.com/imagename:tag`) in the Pod definition.

All users of the cluster who can create pods will be able to run pods that use any of the images in the ECR registry.

The kubelet will fetch and periodically refresh ECR credentials. It needs the following permissions to do this:

- `ecr:GetAuthorizationToken`
- `ecr:BatchCheckLayerAvailability`
- `ecr:GetDownloadUrlForLayer`
- `ecr:GetRepositoryPolicy`
- `ecr:DescribeRepositories`
- `ecr:ListImages`
- `ecr:BatchGetImage`

Requirements:

- You must be using kubelet version `v1.2.0` or newer. (e.g. run `/usr/bin/kubelet --version=true`).
- If your nodes are in region A and your registry is in a different region B, you need version `v1.3.0` or newer.
- ECR must be offered in your region

Troubleshooting:

- Verify all requirements above.
- Get $REGION (e.g. `us-west-2`) credentials on your workstation. SSH into the host and run Docker manually with those creds. Does it work?
- Verify kubelet is running with `--cloud-provider=aws`.
- Check kubelet logs (e.g. `journalctl -u kubelet`) for log lines like:
  - `plugins.go:56] Registering credential provider: aws-ecr-key`
  - `provider.go:91] Refreshing cache for provider: *aws_credentials.ecrProvider`

Using Azure Container Registry (ACR)

When using Azure Container Registry you can authenticate using either an admin user or a service principal. In either case, authentication is done via standard Docker authentication. These instructions assume the azure-cli command line tool.

You first need to create a registry and generate credentials, complete documentation for this can be found in the Azure container registry documentation.

Once you have created your container registry, you will use the following credentials to login:

- `DOCKER_USER` : service principal, or admin username
- `DOCKER_PASSWORD`: service principal password, or admin user password
- `DOCKER_REGISTRY_SERVER`: `${some-registry-name}.azurecr.io`
- `DOCKER_EMAIL`: `${some-email-address}`

Once you have those variables filled in you can configure a Kubernetes Secret and use it to deploy a Pod.

Using IBM Cloud Container Registry

IBM Cloud Container Registry provides a multi-tenant private image registry that you can use to safely store and share your Docker images. By default, images in your private registry are scanned by the integrated Vulnerability Advisor to detect security issues and potential vulnerabilities. Users in your IBM Cloud account can access your images, or you can create a token to grant access to registry namespaces.

To install the IBM Cloud Container Registry CLI plug-in and create a namespace for your images, see Getting started with IBM Cloud Container Registry.

You can use the IBM Cloud Container Registry to deploy containers from IBM Cloud public images and your private images into the `default` namespace of your IBM Cloud Kubernetes Service cluster. To deploy a container into other namespaces, or to use an image from a different IBM Cloud Container Registry region or IBM Cloud account, create a Kubernetes `imagePullSecret`. For more information, see Building containers from images.

Configuring Nodes to Authenticate to a Private Registry

> **Note:** If you are running on Google Kubernetes Engine, there will already be a `.dockercfg` on each node with credentials for Google Container Registry. You cannot use this approach.
>
> **Note:** If you are running on AWS EC2 and are using the EC2 Container Registry (ECR), the kubelet on each node will manage and update the ECR login credentials. You cannot use this approach.
>
> **Note:** This approach is suitable if you can control node configuration. It will not work reliably on GCE, and any other cloud provider that does automatic node replacement.

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 your environment file for kubelet.

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 `$HOME/.docker/config.json`.
2. View `$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=$(kubectl get nodes -o jsonpath='{range.items[*].metadata}{.name} {end}')`
   - if you want to get the IPs: `nodes=$(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: `for n in $nodes; do scp ~/.docker/config.json root@$n:/var/lib/kubelet/config.json; done`

Verify by creating a pod that uses a private image, e.g.:

```
kubectl create -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 should see:

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

If it failed, then you will see:

```
kubectl describe pods/private-image-test-1 | grep "Failed"
  Fri, 26 Jun 2015 15:36:13 -0700    Fri, 26 Jun 2015 15:39:13 -0700    19    {kubelet node-
```

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-pulling Images

> **Note:** If you are running on Google Kubernetes Engine, there will already be a `.dockercfg` on each node with credentials for Google Container Registry. You cannot use this approach.
>
> **Note:** This approach is suitable if you can control node configuration. It will not work reliably on GCE, and any other cloud provider that does automatic node replacement.

By default, the kubelet will try 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 approach is currently the recommended approach for Google Kubernetes Engine, GCE, and any cloud-providers where node creation is automated.

Kubernetes supports specifying 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 myregistrykey --docker-server=DOCKER_REGISTRY_SERVER -
secret/myregistrykey created.
```

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` when pulling images for your Pods.

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

Bypassing kubectl create secrets

If for some reason you need multiple items in a single `.docker/config.json` or need control not given by the above command, then you can create a secret using json or yaml.

Be sure to:

- set the name of the data item to `.dockerconfigjson`
- base64 encode the docker file and paste that string, unbroken as the value for field `data[".dockerconfigjson"]`
- set `type` to `kubernetes.io/dockerconfigjson`

Example:

```
apiVersion: v1
kind: Secret
metadata:
  name: myregistrykey
  namespace: awesomeapps
data:
  .dockerconfigjson: UmVhbGx5IHJlYWxseSByZWVlZWVlZWFhYWFhYWFhYWFhYWFhYWFhYWFhYWFhYWx
type: kubernetes.io/dockerconfigjson
```

If you get the error message `error: no objects passed to create`, it may mean the base64 encoded string is invalid. If you get an error message like `Secret "myregistrykey" is invalid: data[.dockerconfigjson]: invalid value ...`, it means the data was successfully un-base64 encoded, but could not be parsed as a `.docker/config.json` file.

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.

```
apiVersion: v1
kind: Pod
metadata:
  name: foo
  namespace: awesomeapps
spec:
  containers:
    - name: foo
      image: janedoe/awesomeapp:v1
  imagePullSecrets:
    - name: myregistrykey
```

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. This approach will work on Google Kubernetes Engine.

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.
     - On GCE/Google Kubernetes Engine, a local mirror is automatically used for improved speed and availability.
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.
   - Or, when on GCE/Google Kubernetes Engine, use the project's Google Container Registry.
     - 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 a 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.

Edit This Page

Container Environment Variables

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

- Container environment
- What's next

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.

What's next

- Learn more about Container lifecycle hooks.
- Get hands-on experience attaching handlers to Container lifecycle events.

Edit This Page

Runtime Class

FEATURE STATE: `Kubernetes v1.12` alpha

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

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

This page describes the RuntimeClass resource and runtime selection mechanism.

- Runtime Class

Runtime Class

RuntimeClass is an alpha feature for selecting the container runtime configuration to use to run a pod's containers.

Set Up

As an early alpha feature, there are some additional setup steps that must be taken in order to use the RuntimeClass feature:

1. Enable the RuntimeClass feature gate (on apiservers & kubelets, requires version 1.12+)
2. Install the RuntimeClass CRD
3. Configure the CRI implementation on nodes (runtime dependent)
4. Create the corresponding RuntimeClass resources

1. Enable the RuntimeClass feature gate

See Feature Gates for an explanation of enabling feature gates. The `RuntimeClass` feature gate must be enabled on apiservers *and* kubelets.

2. Install the RuntimeClass CRD

The RuntimeClass CustomResourceDefinition (CRD) can be found in the addons directory of the Kubernetes git repo: kubernetes/cluster/addons/runtimeclass/runtimeclass_crd.yaml

Install the CRD with `kubectl apply -f runtimeclass_crd.yaml`.

3. Configure the CRI implementation on nodes

The configurations to select between with RuntimeClass are CRI implementation dependent. See the corresponding documentation for your CRI implementation for how to configure. As this is an alpha feature, not all CRIs support multiple RuntimeClasses yet.

> **Note:** RuntimeClass currently assumes a homogeneous node configuration across the cluster (which means that all nodes are configured the same way with respect to container runtimes). Any heterogeneity (varying configurations) must be managed independently of RuntimeClass through scheduling features (see Assigning Pods to Nodes).

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

4. Create the corresponding RuntimeClass resources

The configurations setup in step 3 should each have an associated `RuntimeHandler` name, which identifies the configuration. For each RuntimeHandler (and optionally the empty `""` handler), create a corresponding RuntimeClass object.

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

```
apiVersion: node.k8s.io/v1alpha1  # 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
spec:
  runtimeHandler: myconfiguration  # The name of the corresponding CRI configuration
```

> **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.

Edit This Page

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
- Container hooks
- What's next

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

`PreStop`

This hook is called immediately before a container is terminated. It is blocking, meaning it is synchronous, so it must complete before the call to delete 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 executes the handler in the Container registered for that hook.

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.

The behavior is similar for a `PreStop` hook. If the hook hangs during execution, the Pod phase stays in a `Terminating` state and is killed after `terminationGracePeriodSeconds` of pod ends. If 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    From                                SubobjectPath        Typ
  ---------    --------    -----    ----                                -------------        ---
  1m        1m        1    {default-scheduler }                                    Normal
  1m        1m        1    {kubelet gke-test-cluster-default-pool-a07e5d30-siqd}    spec.cor
  1m        1m        1    {kubelet gke-test-cluster-default-pool-a07e5d30-siqd}    spec.cor
  1m        1m        1    {kubelet gke-test-cluster-default-pool-a07e5d30-siqd}    spec.cor
  1m        1m        1    {kubelet gke-test-cluster-default-pool-a07e5d30-siqd}    spec.cor
  38s        38s        1    {kubelet gke-test-cluster-default-pool-a07e5d30-siqd}    spec.c
  37s        37s        1    {kubelet gke-test-cluster-default-pool-a07e5d30-siqd}    spec.c
  38s        37s        2    {kubelet gke-test-cluster-default-pool-a07e5d30-siqd}
  1m         22s         2     {kubelet gke-test-cluster-default-pool-a07e5d30-siqd}    spec
```

What's next

- Learn more about the Container environment.
- Get hands-on experience attaching handlers to Container lifecycle events.

Edit This Page

ReplicaSet

ReplicaSet is the next-generation Replication Controller. The only difference between a *ReplicaSet* and a *Replication Controller* right now is the selector support. ReplicaSet supports the new set-based selector requirements as described in the labels user guide whereas a Replication Controller only supports equality-based selector requirements.

- How to use a ReplicaSet
- When to use a ReplicaSet
- Example
- Writing a ReplicaSet Spec
- Working with ReplicaSets
- Alternatives to ReplicaSet

How to use a ReplicaSet

Most `kubectl` commands that support Replication Controllers also support ReplicaSets. One exception is the `rolling-update` command. If you want the rolling update functionality please consider using Deployments instead. Also, the `rolling-update` command is imperative whereas Deployments are declarative, so we recommend using Deployments through the `rollout` command.

While ReplicaSets can be used independently, today it's 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.

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
    matchExpressions:
      - {key: tier, operator: In, values: [frontend]}
  template:
    metadata:
      labels:
        app: guestbook
        tier: frontend
    spec:
      containers:
      - name: php-redis
        image: gcr.io/google_samples/gb-frontend:v3
        resources:
          requests:
            cpu: 100m
            memory: 100Mi
        env:
        - name: GET_HOSTS_FROM
          value: dns
          # If your cluster config does not include a dns service, then to
          # instead access environment variables to find service host
          # info, comment out the 'value: dns' line above, and uncomment the
          # line below.
          # value: env
        ports:
        - containerPort: 80
```

Saving this manifest into `frontend.yaml` and submitting it to a Kubernetes cluster should create the defined ReplicaSet and the pods that it manages.

```
$ kubectl create -f http://k8s.io/examples/controllers/frontend.yaml
replicaset.apps/frontend created
$ kubectl describe rs/frontend
Name:       frontend
Namespace:  default
Selector:   tier=frontend,tier in (frontend)
Labels:     app=guestbook
        tier=frontend
Annotations:    <none>
Replicas:   3 current / 3 desired
Pods Status:    3 Running / 0 Waiting / 0 Succeeded / 0 Failed
Pod Template:
  Labels:       app=guestbook
                tier=frontend
  Containers:
   php-redis:
    Image:      gcr.io/google_samples/gb-frontend:v3
    Port:       80/TCP
    Requests:
      cpu:      100m
      memory:   100Mi
    Environment:
      GET_HOSTS_FROM:   dns
    Mounts:             <none>
  Volumes:              <none>
Events:
  FirstSeen    LastSeen    Count    From                SubobjectPath    Type        Reason
  ---------    --------    -----    ----                -------------    --------    ------
  1m           1m          1        {replicaset-controller }             Normal      Success
  1m           1m          1        {replicaset-controller }             Normal      Success
  1m           1m          1        {replicaset-controller }             Normal      Success
$ kubectl get pods
NAME             READY     STATUS    RESTARTS   AGE
frontend-9si5l   1/1       Running   0          1m
frontend-dnjpy   1/1       Running   0          1m
frontend-qhloh   1/1       Running   0          1m
```

Writing a ReplicaSet Spec

As with all other Kubernetes API objects, a ReplicaSet needs the `apiVersion`, `kind`, and `metadata` fields. For general information about working with manifests, see object management using kubectl.

A ReplicaSet 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 that it is nested and does not have an `apiVersion` or `kind`.

In addition to required fields of a pod, a pod template in a ReplicaSet must specify appropriate labels and an appropriate restart policy.

For labels, make sure to not overlap with other controllers. For more information, see pod selector.

For restart policy, the only allowed value for `.spec.template.spec.restartPolicy` is `Always`, which is the default.

For local container restarts, ReplicaSet delegates to an agent on the node, for example the Kubelet or Docker.

Pod Selector

The `.spec.selector` field is a label selector. A ReplicaSet 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 ReplicaSet to be replaced without affecting the running pods.

The `.spec.template.metadata.labels` must match the `.spec.selector`, or it will be rejected by the API.

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.

Also you should not normally create any pods whose labels match this selector, either directly, with another ReplicaSet, or with another controller such as a Deployment. If you do so, the ReplicaSet 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.

Labels on a ReplicaSet

The ReplicaSet can itself have labels (`.metadata.labels`). Typically, you would set these the same as the `.spec.template.metadata.labels`. However, they are allowed to be different, and the `.metadata.labels` do not affect the behavior of the ReplicaSet.

Replicas

You can specify how many pods should run concurrently by setting `.spec.replicas`. 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 shut down, and a replacement starts early.

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 delete option. e.g. :

```
kubectl proxy --port=8080
curl -X DELETE  'localhost:8080/apis/extensions/v1beta1/namespaces/default/replicasets/front
> -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`, e.g. :

```
kubectl proxy --port=8080
curl -X DELETE  'localhost:8080/apis/extensions/v1beta1/namespaces/default/replicasets/front
> -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 rolling update.

Isolating pods from a ReplicaSet

Pods may be removed from a ReplicaSet'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).

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 an 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 create -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
```

Alternatives to ReplicaSet

Deployment (Recommended)

`Deployment` is a higher-level API object that updates its underlying ReplicaSets and their Pods in a similar fashion as `kubectl rolling-update`. Deployments are recommended if you want this rolling update functionality, because unlike `kubectl rolling-update`, they are declarative, server-side, and have additional features. For more information on running a stateless application using a Deployment, please read Run a Stateless Application Using a Deployment.

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.

Edit This Page

Pod Preset

This page provides an overview of PodPresets, which are objects for injecting certain information into pods at creation time. The information can include secrets, volumes, volume mounts, and environment variables.

- Understanding Pod Presets
- How It Works
- Enable Pod Preset
- What's next

Understanding Pod Presets

A `Pod Preset` is an API resource for injecting additional runtime requirements into a Pod at creation time. You use label selectors to specify the Pods to which a given Pod Preset applies.

Using a Pod Preset allows pod template authors to not have to explicitly provide all information for every pod. This way, authors of pod templates consuming a specific service do not need to know all the details about that service.

For more information about the background, see the design proposal for PodPreset.

How It Works

Kubernetes provides an admission controller (`PodPreset`) which, when enabled, applies Pod Presets to incoming pod creation requests. When a pod creation request occurs, the system does the following:

1. Retrieve all `PodPresets` available for use.
2. Check if the label selectors of any `PodPreset` matches the labels on the pod being created.
3. Attempt to merge the various resources defined by the `PodPreset` into the Pod being created.
4. On error, throw an event documenting the merge error on the pod, and create the pod *without* any injected resources from the `PodPreset`.
5. Annotate the resulting modified Pod spec to indicate that it has been modified by a `PodPreset`. The annotation is of the form `podpreset.admission.kubernetes.io/podpreset-<pod-preset name>: "<resource version>"`.

Each Pod can be matched by zero or more Pod Presets; and each `PodPreset` can be applied to zero or more pods. When a `PodPreset` is applied to one or more Pods, Kubernetes modifies the Pod Spec. For changes to `Env`, `EnvFrom`, and `VolumeMounts`, Kubernetes modifies the container spec for all containers in the Pod; for changes to `Volume`, Kubernetes modifies the Pod Spec.

> **Note:** A Pod Preset is capable of modifying the `.spec.containers` field in a Pod spec when appropriate. *No* resource definition from the Pod Preset will be applied to the `initContainers` field.

Disable Pod Preset for a Specific Pod

There may be instances where you wish for a Pod to not be altered by any Pod Preset mutations. In these cases, you can add an annotation in the Pod Spec of the form: `podpreset.admission.kubernetes.io/exclude: "true"`.

Enable Pod Preset

In order to use Pod Presets in your cluster you must ensure the following:

1. You have enabled the API type `settings.k8s.io/v1alpha1/podpreset`. For example, this can be done by including `settings.k8s.io/v1alpha1=true` in the `--runtime-config` option for the API server.
2. You have enabled the admission controller `PodPreset`. One way to doing this is to include `PodPreset` in the `--enable-admission-plugins` option value specified for the API server.
3. You have defined your Pod Presets by creating `PodPreset` objects in the namespace you will use.

What's next

- Injecting data into a Pod using PodPreset

Edit This Page

Pod Overview

This page provides an overview of `Pod`, the smallest deployable object in the Kubernetes object model.

- Understanding Pods
- Working with Pods
- Pod Templates
- What's next

Understanding Pods

A *Pod* is the basic building block of Kubernetes–the smallest and simplest unit in the Kubernetes object model that you create or deploy. A Pod represents a running process on your cluster.

A Pod encapsulates an application container (or, in some cases, multiple containers), storage resources, a unique network IP, and options that govern how

the container(s) should run. A Pod represents a unit of deployment: *a single instance of an application in Kubernetes*, which might consist of either a single container or a small number of containers that are tightly coupled and that share resources.

> Docker is the most common container runtime used in a Kubernetes Pod, but Pods support other container runtimes as well.

Pods in a Kubernetes cluster can be 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, and Kubernetes manages the Pods rather than the containers directly.
- **Pods that run multiple containers that need to work together**. A Pod might encapsulate an application composed of multiple co-located containers that are tightly coupled and need to share resources. These co-located containers might form a single cohesive unit of service–one container serving files from a shared volume to the public, while a separate "sidecar" container refreshes or updates those files. The Pod wraps these containers and storage resources together as a single manageable entity.

The Kubernetes Blog has some additional information on Pod use cases. For more information, see:

- The Distributed System Toolkit: Patterns for Composite Containers
- Container Design Patterns

Each Pod is meant to run a single instance of a given application. If you want to scale your application horizontally (e.g., run multiple instances), you should use multiple Pods, one for each instance. In Kubernetes, this is generally referred to as *replication*. Replicated Pods are usually created and managed as a group by an abstraction called a Controller. See Pods and Controllers for more information.

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.

Note that 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. 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:

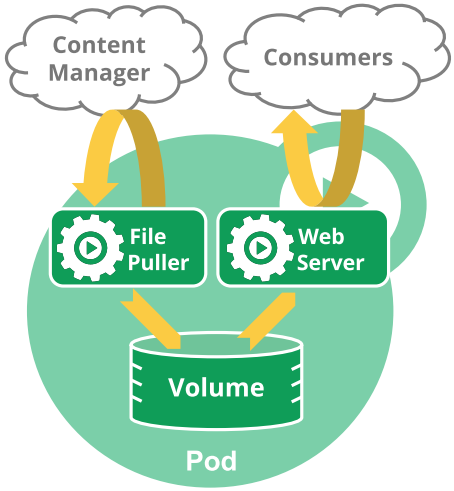


Figure 1: pod diagram

Pods provide two kinds of shared resources for their constituent containers: *networking* and *storage*.

Networking

Each Pod is assigned a unique IP address. Every container in a Pod shares the network namespace, including the IP address and network ports. Containers *inside a 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).

Storage

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 Volumes for more information on how Kubernetes implements shared storage in a Pod.

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), it is scheduled to run on a Node in your cluster. The Pod remains on that Node until the process is terminated, 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 the Pod. The Pod itself does not run, but is an environment the containers run in and persists until it is deleted.

Pods do not, by themselves, self-heal. If a Pod is scheduled to a Node that 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. Thus, while it is possible to use Pod directly, it's far more common in Kubernetes to manage your pods using a Controller. See Pods and Controllers for more information on how Kubernetes uses Controllers to implement Pod scaling and healing.

Pods and Controllers

A Controller can create and manage multiple Pods for you, handling replication and rollout and providing self-healing capabilities at cluster scope. For example, if a Node fails, the Controller might automatically replace the Pod by scheduling an identical replacement on a different Node.

Some examples of Controllers that contain one or more pods include:

- Deployment
- StatefulSet
- DaemonSet

In general, Controllers use a Pod Template that you provide to create the Pods for which it is responsible.

Pod Templates

Pod templates are pod specifications which are included in other objects, such as Replication Controllers, Jobs, and DaemonSets. Controllers use Pod Templates to make actual pods. The sample below is a simple manifest for a Pod which contains a container that prints a message.

```
apiVersion: v1
kind: Pod
metadata:
  name: myapp-pod
  labels:
    app: myapp
```

```
spec:
  containers:
  - name: myapp-container
    image: busybox
    command: ['sh', '-c', 'echo Hello Kubernetes! && sleep 3600']
```

Rather than specifying the current desired state of all replicas, pod templates are like cookie cutters. Once a cookie has been cut, the cookie has no relationship to the cutter. There is no "quantum entanglement". Subsequent changes to the template or even switching to a new template has no direct effect on the pods already created. Similarly, pods created by a replication controller may subsequently be updated directly. This is in deliberate contrast to pods, which do specify the current desired state of all containers belonging to the pod. This approach radically simplifies system semantics and increases the flexibility of the primitive.

What's next

- Learn more about Pod behavior:
  - Pod Termination
  - Other Pod Topics

Edit This Page

Pods

Pods are the smallest deployable units of computing that can be created and managed in Kubernetes.

- What is a Pod?
- Motivation for pods
- Uses of pods
- Alternatives considered
- Durability of pods (or lack thereof)
- Termination of Pods
- Privileged mode for pod containers
- API Object

What is a Pod?

A *pod* (as in a pod of whales or pea pod) is a group of one or more containers (such as Docker containers), with shared storage/network, 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 a pre-container world, being executed on the same physical or virtual machine would mean being executed on the same logical host.

While Kubernetes supports more container runtimes than just Docker, Docker is the most commonly known runtime, and it helps to describe pods in Docker terms.

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.

Containers within a pod share an IP address and port space, and can find each other via `localhost`. They 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. These containers usually communicate with each other via Pod IP addresses.

Applications within a pod also have access to shared volumes, which are defined as part of a pod and are made available to be mounted into each application's filesystem.

In terms of Docker constructs, a pod is modelled as a group of Docker containers with shared namespaces and shared volumes.

Like individual application containers, pods are considered to be relatively ephemeral (rather than durable) entities. As discussed in life of a pod, 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. A given pod (as defined by a UID) is not "rescheduled" to a new node; instead, it can be replaced by an identical pod, with even the same name if desired, but with a new UID (see replication controller for more details).

When something is said to have the same lifetime as a pod, such as a volume, that means that it exists as long as that pod (with that UID) exists. If that pod is deleted for any reason, even if an identical replacement is created, the related thing (e.g. volume) is also destroyed and created anew.

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

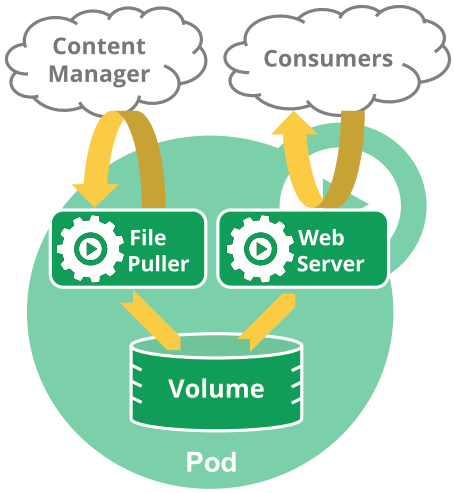


Figure 2: pod diagram

Motivation for pods

Management

Pods are a model of the pattern of multiple cooperating processes which form a cohesive unit of service. They simplify application deployment and management by providing a higher-level abstraction than the set of their constituent applications. Pods serve as unit of deployment, horizontal scaling, and replication. Colocation (co-scheduling), shared fate (e.g. termination), coordinated replication, resource sharing, and dependency management are handled automatically for containers in a pod.

Resource sharing and communication

Pods enable data sharing and communication among their constituents.

The applications in a pod all use the same network namespace (same IP and port space), and can thus "find" each other and communicate using `localhost`. Because of this, applications in a pod must coordinate their usage of ports. Each pod has an IP address in a flat shared networking space that has full communication with other physical computers and pods across the network.

The hostname is set to the pod's Name for the application containers within the pod. More details on networking.

In addition to defining the application containers that run in the pod, the pod

specifies a set of shared storage volumes. Volumes enable data to survive container restarts and to be shared among the applications within the pod.

Uses of pods

Pods can be used to host vertically integrated application stacks (e.g. LAMP), but their primary motivation is to support co-located, co-managed helper programs, such as:

- content management systems, file and data loaders, local cache managers, etc.
- log and checkpoint backup, compression, rotation, snapshotting, etc.
- data change watchers, log tailers, logging and monitoring adapters, event publishers, etc.
- proxies, bridges, and adapters
- controllers, managers, configurators, and updaters

Individual pods are not intended to run multiple instances of the same application, in general.

For a longer explanation, see The Distributed System ToolKit: Patterns for Composite Containers.

Alternatives considered

Why not just run multiple programs in a single (Docker) container?

1. Transparency. Making the containers within the pod visible to the infrastructure enables the infrastructure to provide services to those containers, such as process management and resource monitoring. This facilitates a number of conveniences for users.
2. Decoupling software dependencies. The individual containers may be versioned, rebuilt and redeployed independently. Kubernetes may even support live updates of individual containers someday.
3. Ease of use. Users don't need to run their own process managers, worry about signal and exit-code propagation, etc.
4. Efficiency. Because the infrastructure takes on more responsibility, containers can be lighter weight.

Why not support affinity-based co-scheduling of containers?

That approach would provide co-location, but would not provide most of the benefits of pods, such as resource sharing, IPC, guaranteed fate sharing, and simplified management.

Durability of pods (or lack thereof)

Pods aren't intended to be treated as durable entities. They won't survive scheduling failures, node failures, or other evictions, such as due to lack of resources, or in the case of node maintenance.

In general, users shouldn't need to create pods directly. They should almost always use controllers even for singletons, for example, Deployments. Controllers provide self-healing with a cluster scope, as well as replication and rollout management. Controllers like StatefulSet can also provide support to stateful pods.

The use of collective APIs as the primary user-facing primitive is relatively common among cluster scheduling systems, including Borg, Marathon, Aurora, and Tupperware.

Pod is exposed as a primitive in order to facilitate:

- scheduler and controller pluggability
- support for pod-level operations without the need to "proxy" them via controller APIs
- decoupling of pod lifetime from controller lifetime, such as for bootstrapping
- decoupling of controllers and services — the endpoint controller just watches pods
- clean composition of Kubelet-level functionality with cluster-level functionality — Kubelet is effectively the "pod controller"
- high-availability applications, which will expect pods to be replaced in advance of their termination and certainly in advance of deletion, such as in the case of planned evictions or image prefetching.

Termination of Pods

Because pods represent running processes on nodes in the cluster, it is important to allow those processes to gracefully terminate when they are no longer needed (vs being violently killed with a KILL signal and having no chance to clean up). Users should be able to request deletion and know when processes terminate, but also be able to ensure that deletes eventually complete. When a user requests deletion of a pod, the system records the intended grace period before the pod is allowed to be forcefully killed, and a TERM signal is sent to the main process in each container. Once the grace period has expired, the KILL signal is sent to those processes, and the pod is then deleted from the API server. If the Kubelet or the container manager is restarted while waiting for processes to terminate, the termination will be retried with the full grace period.

An example flow:

1. User sends command to delete Pod, with default grace period (30s)

2. The Pod in the API server is updated with the time beyond which the Pod is considered "dead" along with the grace period.
3. Pod shows up as "Terminating" when listed in client commands
4. (simultaneous with 3) When the Kubelet sees that a Pod has been marked as terminating because the time in 2 has been set, it begins the pod shutdown process.
   (a) If the pod has defined a preStop hook, it is invoked inside of the pod. If the `preStop` hook is still running after the grace period expires, step 2 is then invoked with a small (2 second) extended grace period.
   (b) The processes in the Pod are sent the TERM signal.
5. (simultaneous with 3) Pod is removed from endpoints list for service, and are no longer considered part of the set of running pods for replication controllers. Pods that shutdown slowly cannot continue to serve traffic as load balancers (like the service proxy) remove them from their rotations.
6. When the grace period expires, any processes still running in the Pod are killed with SIGKILL.
7. The Kubelet will finish deleting the Pod on the API server by setting grace period 0 (immediate deletion). The Pod disappears from the API and is no longer visible from the client.

By default, all deletes are graceful within 30 seconds. The `kubectl delete` command supports the `--grace-period=<seconds>` option which allows a user to override the default and specify their own value. The value `0` force deletes the pod. In kubectl version >= 1.5, you must specify an additional flag `--force` along with `--grace-period=0` in order to perform force deletions.

Force deletion of pods

Force deletion of a pod is defined as deletion of a pod from the cluster state and etcd immediately. When a force deletion is performed, the apiserver 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.

Force deletions can be potentially dangerous for some pods and should be performed with caution. In case of StatefulSet pods, please refer to the task documentation for deleting Pods from a StatefulSet.

Privileged mode for pod containers

From Kubernetes v1.1, any container in a pod can enable privileged mode, using the `privileged` flag on the `SecurityContext` of the container spec. This is useful for containers that want to use linux capabilities like manipulating the

network stack and accessing devices. Processes within the container get almost the same privileges that are available to processes outside a container. With privileged mode, it should be easier to write network and volume plugins as separate pods that don't need to be compiled into the kubelet.

If the master is running Kubernetes v1.1 or higher, and the nodes are running a version lower than v1.1, then new privileged pods will be accepted by api-server, but will not be launched. They will be pending state. If user calls `kubectl describe pod FooPodName`, user can see the reason why the pod is in pending state. The events table in the describe command output will say: `Error validating pod "FooPodName"."FooPodNamespace" from api, ignoring: spec.containers[0].securityContext.privileged: forbidden '<*>(0xc2089d3248)true'`

If the master is running a version lower than v1.1, then privileged pods cannot be created. If user attempts to create a pod, that has a privileged container, the user will get the following error: `The Pod "FooPodName" is invalid. spec.containers[0].securityContext.privileged: forbidden '<*>(0xc20b222db0)true'`

API Object

Pod is a top-level resource in the Kubernetes REST API. More details about the API object can be found at: Pod API object.

Edit This Page

Pod Lifecycle

This page describes the lifecycle of a Pod.

- Pod phase
- Pod conditions
- Container probes
- Pod and Container status
- Pod readiness gate
- Restart policy
- Pod lifetime
- Examples
- What's next

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 system, but one or more of the Container images
`Running`	The Pod has been bound to a node, and all of the Containers have been created. At least one C
`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
`Unknown`	For some reason the state of the Pod could not be obtained, typically due to an error in commu

Pod conditions

A Pod has a PodStatus, which has an array of PodConditions through which the Pod has or has not passed. Each element of the PodCondition array has six possible fields:

- The `lastProbeTime` field provides a timestamp for when the Pod condition was last probed.
- The `lastTransitionTime` field provides a timestamp for when the Pod last transitioned from one status to another.
- The `message` field is a human-readable message indicating details about the transition.
- The `reason` field is a unique, one-word, CamelCase reason for the condition's last transition.
- The `status` field is a string, with possible values "`True`", "`False`", and "`Unknown`".
- The `type` field is a string with the following possible values:
  - `PodScheduled`: the Pod has been scheduled to a node;
  - `Ready`: the Pod is able to serve requests and should be added to the load balancing pools of all matching Services;
  - `Initialized`: all init containers have started successfully;
  - `Unschedulable`: the scheduler cannot schedule the Pod right now, for example due to lacking of resources or other constraints;
  - `ContainersReady`: all containers in the Pod are ready.

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 Container'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 Container'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 two 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 service 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`.

When should you use liveness or readiness probes?

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.

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 that 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.

For more information about how to set up a liveness or readiness probe, see Configure Liveness and Readiness Probes.

Pod and Container status

For detailed information about Pod Container status, see PodStatus and ContainerStatus. Note that the information reported as Pod status depends on the current ContainerState.

Pod readiness gate

FEATURE STATE: `Kubernetes v1.12` beta

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

- The version names contain beta (e.g. v2beta3).
- Code is well tested. Enabling the feature is considered safe. Enabled by default.
- Support for the overall feature will not be dropped, though details may change.
- The schema and/or semantics of objects may change in incompatible ways in a subsequent beta or stable release. When this happens, we will provide instructions for migrating to the next version. This may require deleting, editing, and re-creating API objects. The editing process may require some thought. This may require downtime for applications that rely on the feature.

- Recommended for only non-business-critical uses because of potential for incompatible changes in subsequent releases. If you have multiple clusters that can be upgraded independently, you may be able to relax this restriction.
- **Please do try our beta features and give feedback on them! After they exit beta, it may not be practical for us to make more changes.**

In order to add extensibility to Pod readiness by enabling the injection of extra feedbacks or signals into `PodStatus`, Kubernetes 1.11 introduced a feature named Pod ready++. You can use the new field `ReadinessGate` in the `PodSpec` to specify additional conditions to be evaluated for Pod readiness. If Kubernetes cannot find such a condition in the `status.conditions` field of a Pod, the status of the condition is default to "`False`". Below is an example:

```
Kind: Pod
...
spec:
  readinessGates:
    - conditionType: "www.example.com/feature-1"
status:
  conditions:
    - type: Ready  # this is a builtin PodCondition
      status: "True"
      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 new Pod conditions must comply with Kubernetes label key format. Since the `kubectl patch` command still doesn't support patching object status, the new Pod conditions have to be injected through the `PATCH` action using one of the KubeClient libraries.

With the introduction of new Pod conditions, a Pod is evaluated to be ready **only** when both the following statements are true:

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

To facilitate this change to Pod readiness evaluation, a new Pod condition `ContainersReady` is introduced to capture the old Pod `Ready` condition.

In K8s 1.11, as an alpha feature, the "Pod Ready++" feature has to be explicitly enabled by setting the `PodReadinessGates` feature gate to true.

In K8s 1.12, the feature is enabled by default.

Restart policy

A PodSpec has a `restartPolicy` field with possible values Always, OnFailure, and Never. The default value is Always. `restartPolicy` applies to all Containers in the Pod. `restartPolicy` only refers to restarts of the Containers by the kubelet on the same node. Exited Containers that are restarted by the kubelet are restarted with an exponential back-off delay (10s, 20s, 40s …) capped at five minutes, and is reset after ten minutes of successful execution. As discussed in the Pods document, once bound to a node, a Pod will never be rebound to another node.

Pod lifetime

In general, Pods do not disappear until someone destroys them. This might be a human or a controller. The only exception to this rule is that Pods with a `phase` of Succeeded or Failed for more than some duration (determined by `terminated-pod-gc-threshold` in the master) will expire and be automatically destroyed.

Three types of controllers are available:

- Use a Job for Pods that are expected to terminate, for example, batch computations. Jobs are appropriate only for Pods with `restartPolicy` equal to OnFailure or Never.
- Use a ReplicationController, ReplicaSet, or Deployment for Pods that are not expected to terminate, for example, web servers. ReplicationControllers are appropriate only for Pods with a `restartPolicy` of Always.
- Use a DaemonSet for Pods that need to run one per machine, because they provide a machine-specific system service.

All three types of controllers contain a PodTemplate. It is recommended to create the appropriate controller and let it create Pods, rather than directly create Pods yourself. That is because Pods alone are not resilient to machine failures, but controllers are.

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.

Examples

Advanced liveness probe example

Liveness probes are executed by the kubelet, so all requests are made in the kubelet network namespace.

```
apiVersion: v1
kind: Pod
metadata:
  labels:
    test: liveness
  name: liveness-http
spec:
  containers:
  - args:
    - /server
    image: k8s.gcr.io/liveness
    livenessProbe:
      httpGet:
        # when "host" is not defined, "PodIP" will be used
        # host: my-host
        # when "scheme" is not defined, "HTTP" scheme will be used. Only "HTTP" and "HTTPS"
        # scheme: HTTPS
        path: /healthz
        port: 8080
        httpHeaders:
        - name: X-Custom-Header
          value: Awesome
      initialDelaySeconds: 15
      timeoutSeconds: 1
    name: liveness
```

Example states

- Pod is running and has one Container. Container exits with success.
  - Log completion event.
  - If `restartPolicy` is:
    - Always: Restart Container; Pod `phase` stays Running.
    - OnFailure: Pod `phase` becomes Succeeded.
    - Never: Pod `phase` becomes Succeeded.
- Pod is running and has one Container. Container exits with failure.
  - Log failure event.
  - If `restartPolicy` is:

        * Always: Restart Container; Pod `phase` stays Running.
        * OnFailure: Restart Container; Pod `phase` stays Running.
        * Never: Pod `phase` becomes Failed.

- Pod is running and has two Containers. Container 1 exits with failure.
  - Log failure event.
  - If `restartPolicy` is:
    * Always: Restart Container; Pod `phase` stays Running.
    * OnFailure: Restart Container; Pod `phase` stays Running.
    * Never: Do not restart Container; Pod `phase` stays Running.
  - If Container 1 is not running, and Container 2 exits:
    * Log failure event.
    * If `restartPolicy` is:
      · Always: Restart Container; Pod `phase` stays Running.
      · OnFailure: Restart Container; Pod `phase` stays Running.
      · Never: Pod `phase` becomes Failed.

- Pod is running and has one Container. Container runs out of memory.
  - Container terminates in failure.
  - Log OOM event.
  - If `restartPolicy` is:
    * Always: Restart Container; Pod `phase` stays Running.
    * OnFailure: Restart Container; Pod `phase` stays Running.
    * Never: Log failure event; Pod `phase` becomes Failed.

- Pod is running, and a disk dies.
  - Kill all Containers.
  - Log appropriate event.
  - Pod `phase` becomes Failed.
  - If running under a controller, Pod is recreated elsewhere.

- Pod is running, and its node is segmented out.
  - Node controller waits for timeout.
  - Node controller sets Pod `phase` to Failed.
  - If running under a controller, Pod is recreated elsewhere.

What's next

- Get hands-on experience attaching handlers to Container lifecycle events.
- Get hands-on experience configuring liveness and readiness probes.
- Learn more about Container lifecycle hooks.

Edit This Page

Init Containers

This page provides an overview of Init Containers, which are specialized Containers that run before app Containers and can contain utilities or setup scripts not present in an app image.

- Understanding Init Containers
- What can Init Containers be used for?
- Detailed behavior
- Support and compatibility
- What's next

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:

- They always run to completion.
- Each one must complete successfully before the next one is started.

If an Init Container fails for a Pod, Kubernetes restarts the Pod repeatedly until the Init Container succeeds. However, if the Pod has a `restartPolicy` of Never, it is not restarted.

To specify a Container as an Init Container, add the `initContainers` field on the PodSpec as a JSON 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 slightly differently, which are documented in Resources below. Also, Init Containers do not support readiness probes because they must run to completion before the Pod can be ready.

If multiple Init Containers are specified for a Pod, those Containers are run one at a time in sequential order. Each must succeed before the next can run. When all of the Init Containers have run to completion, Kubernetes initializes the Pod and runs the application Containers as usual.

What can Init Containers be used for?

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

- They can contain and run utilities that are not desirable to include in the app Container image for security reasons.
- They can contain utilities or custom code for setup that is 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.
- They use Linux namespaces so that they have different filesystem views from app Containers. Consequently, they can be given access to Secrets that app Containers are not able to access.
- They run to completion before any app Containers start, whereas app Containers run in parallel, so Init Containers provide an easy way to block or delay the startup of app Containers until some set of preconditions are met.

Examples

Here are some ideas for how to use Init Containers:

- Wait for a service to be created with a shell 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/regis
  -d 'instance=$()&ip=$()'

- 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.

More detailed usage examples can be found in the StatefulSets documentation and the Production Pods guide.

Init Containers in use

The following yaml file for Kubernetes 1.5 outlines a simple Pod which has two Init Containers. The first waits for `myservice` and the second waits for `mydb`. Once both containers complete, the Pod will begin.

```
apiVersion: v1
kind: Pod
metadata:
  name: myapp-pod
  labels:
    app: myapp
  annotations:
    pod.beta.kubernetes.io/init-containers: '[
        {
            "name": "init-myservice",
            "image": "busybox",
            "command": ["sh", "-c", "until nslookup myservice; do echo waiting for myservice
        },
        {
            "name": "init-mydb",
            "image": "busybox",
            "command": ["sh", "-c", "until nslookup mydb; do echo waiting for mydb; sleep 2;
        }
    ]'
spec:
  containers:
  - name: myapp-container
    image: busybox
    command: ['sh', '-c', 'echo The app is running! && sleep 3600']
```

There is a new syntax in Kubernetes 1.6, although the old annotation syntax still works for 1.6 and 1.7. The new syntax must be used for 1.8 or greater. We have moved the declaration of Init Containers to `spec`:

```
apiVersion: v1
kind: Pod
metadata:
  name: myapp-pod
  labels:
    app: myapp
spec:
  containers:
  - name: myapp-container
    image: busybox
    command: ['sh', '-c', 'echo The app is running! && sleep 3600']
  initContainers:
```

```
  - name: init-myservice
    image: busybox
    command: ['sh', '-c', 'until nslookup myservice; do echo waiting for myservice; sleep 2;
  - name: init-mydb
    image: busybox
    command: ['sh', '-c', 'until nslookup mydb; do echo waiting for mydb; sleep 2; done;']
```

1.5 syntax still works on 1.6, but we recommend using 1.6 syntax. In Kubernetes 1.6, Init Containers were made a field in the API. The beta annotation is still respected in 1.6 and 1.7, but is not supported in 1.8 or greater.

Yaml file below outlines the `mydb` and `myservice` services:

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

This Pod can be started and debugged with the following commands:

```
$ kubectl create -f myapp.yaml
pod/myapp-pod created
$ kubectl get -f myapp.yaml
NAME        READY     STATUS     RESTARTS   AGE
myapp-pod   0/1       Init:0/2   0          6m
$ 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
  ---------    --------    -----    ----                        -------------
  16s          16s         1        {default-scheduler }
  16s          16s         1        {kubelet 172.17.4.201}    spec.initContainers{init-myser
  13s          13s         1        {kubelet 172.17.4.201}    spec.initContainers{init-myser
  13s          13s         1        {kubelet 172.17.4.201}    spec.initContainers{init-myser
  13s          13s         1        {kubelet 172.17.4.201}    spec.initContainers{init-myser
$ kubectl logs myapp-pod -c init-myservice # Inspect the first init container
$ kubectl logs myapp-pod -c init-mydb      # Inspect the second init container
```

Once we start the `mydb` and `myservice` services, we can see the Init Containers complete and the `myapp-pod` is created:

```
$ kubectl create -f services.yaml
service/myservice created
service/mydb created
$ kubectl get -f myapp.yaml
NAME        READY     STATUS    RESTARTS   AGE
myapp-pod   1/1       Running   0          9m
```

This example is very simple but should provide some inspiration for you to create your own Init Containers.

Detailed behavior

During the startup of a Pod, the Init Containers are started in order, after the network and volumes are initialized. Each Container must exit successfully before the next is started. 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 `Initializing` set to true.

If the Pod 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.
- QoS 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 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 PodSpec causing the Init Container image to change. 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.

Support and compatibility

A cluster with Apiserver version 1.6.0 or greater supports Init Containers using the `.spec.initContainers` field. Previous versions support Init Containers using the alpha or beta annotations. The `.spec.initContainers` field is also mirrored into alpha and beta annotations so that Kubelets version 1.3.0 or greater can execute Init Containers, and so that a version 1.6 apiserver can safely be rolled back to version 1.5.x without losing Init Container functionality for existing created pods.

In Apiserver and Kubelet versions 1.8.0 or greater, support for the alpha and beta annotations is removed, requiring a conversion from the deprecated annotations to the `.spec.initContainers` field.

This feature has exited beta in 1.6. Init Containers can be specified in the PodSpec alongside the app `containers` array. The beta annotation value will still be respected and overrides the PodSpec field value, however, they are deprecated in 1.6 and 1.7. In 1.8, the annotations are no longer supported and must be converted to the PodSpec field.

What's next

- Creating a Pod that has an Init Container

Edit This Page

Pod Preset

This page provides an overview of PodPresets, which are objects for injecting certain information into pods at creation time. The information can include secrets, volumes, volume mounts, and environment variables.

- Understanding Pod Presets
- How It Works
- Enable Pod Preset
- What's next

Understanding Pod Presets

A `Pod Preset` is an API resource for injecting additional runtime requirements into a Pod at creation time. You use label selectors to specify the Pods to which a given Pod Preset applies.

Using a Pod Preset allows pod template authors to not have to explicitly provide all information for every pod. This way, authors of pod templates consuming a specific service do not need to know all the details about that service.

For more information about the background, see the design proposal for PodPreset.

How It Works

Kubernetes provides an admission controller (`PodPreset`) which, when enabled, applies Pod Presets to incoming pod creation requests. When a pod creation request occurs, the system does the following:

1. Retrieve all `PodPresets` available for use.
2. Check if the label selectors of any `PodPreset` matches the labels on the pod being created.
3. Attempt to merge the various resources defined by the `PodPreset` into the Pod being created.
4. On error, throw an event documenting the merge error on the pod, and create the pod *without* any injected resources from the `PodPreset`.
5. Annotate the resulting modified Pod spec to indicate that it has been modified by a `PodPreset`. The annotation is of the form `podpreset.admission.kubernetes.io/podpreset-<pod-preset name>: "<resource version>"`.

Each Pod can be matched by zero or more Pod Presets; and each `PodPreset` can be applied to zero or more pods. When a `PodPreset` is applied to one or more Pods, Kubernetes modifies the Pod Spec. For changes to `Env`, `EnvFrom`, and `VolumeMounts`, Kubernetes modifies the container spec for all containers in the Pod; for changes to `Volume`, Kubernetes modifies the Pod Spec.

> **Note:** A Pod Preset is capable of modifying the `.spec.containers` field in a Pod spec when appropriate. *No* resource definition from the Pod Preset will be applied to the `initContainers` field.

Disable Pod Preset for a Specific Pod

There may be instances where you wish for a Pod to not be altered by any Pod Preset mutations. In these cases, you can add an annotation in the Pod Spec of the form: `podpreset.admission.kubernetes.io/exclude: "true"`.

Enable Pod Preset

In order to use Pod Presets in your cluster you must ensure the following:

1. You have enabled the API type `settings.k8s.io/v1alpha1/podpreset`. For example, this can be done by including `settings.k8s.io/v1alpha1=true` in the `--runtime-config` option for the API server.
2. You have enabled the admission controller `PodPreset`. One way to doing this is to include `PodPreset` in the `--enable-admission-plugins` option value specified for the API server.
3. You have defined your Pod Presets by creating `PodPreset` objects in the namespace you will use.

What's next

- Injecting data into a Pod using PodPreset

Edit This Page

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
- Dealing with Disruptions
- How Disruption Budgets Work
- PDB Example
- Separating Cluster Owner and Application Owner Roles
- How to perform Disruptive Actions on your Cluster
- What's next

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.

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.

Kubernetes offers features to help run highly available applications at the same time as frequent voluntary disruptions. We call this set of features *Disruption Budgets*.

How Disruption Budgets Work

An Application Owner can create a `PodDisruptionBudget` object (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 Pod Disruption Budgets by calling the Eviction API instead of directly deleting pods. Examples are the `kubectl drain` command and the Kubernetes-on-GCE cluster upgrade script (`cluster/gce/upgrade.sh`).

When a cluster administrator wants to drain a node they use the `kubectl drain` command. That tool tries to evict all the pods on the machine. The eviction request may be temporarily rejected, and the tool periodically retries all failed requests until all pods 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 pods controller. The controller is discovered from the pods using the `.metadata.ownerReferences` of the object.

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 controllers (like deployment and stateful-set) are not limited by PDBs when doing rolling upgrades – the handling of failures during application updates is configured in the controller spec. (Learn about updating a deployment.)

When a pod is evicted using the eviction API, it is gracefully terminated (see `terminationGracePeriodSeconds` in PodSpec.)

PDB 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-x *available*	pod-b *available*	pod-c *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-x *terminating*	pod-b *available*	pod-c *available*

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-1`,

would need to terminate completely before its replacement, which is also called `pod-1` 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 *available*	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.
- Fail over 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

Edit This Page

ReplicaSet

ReplicaSet is the next-generation Replication Controller. The only difference between a *ReplicaSet* and a *Replication Controller* right now is the selector support. ReplicaSet supports the new set-based selector requirements as described in the labels user guide whereas a Replication Controller only supports equality-based selector requirements.

- How to use a ReplicaSet
- When to use a ReplicaSet
- Example
- Writing a ReplicaSet Spec
- Working with ReplicaSets
- Alternatives to ReplicaSet

How to use a ReplicaSet

Most `kubectl` commands that support Replication Controllers also support ReplicaSets. One exception is the `rolling-update` command. If you want the rolling update functionality please consider using Deployments instead. Also, the `rolling-update` command is imperative whereas Deployments are declarative, so we recommend using Deployments through the `rollout` command.

While ReplicaSets can be used independently, today it's 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.

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
    matchExpressions:
      - {key: tier, operator: In, values: [frontend]}
  template:
    metadata:
      labels:
        app: guestbook
        tier: frontend
    spec:
      containers:
      - name: php-redis
        image: gcr.io/google_samples/gb-frontend:v3
        resources:
          requests:
            cpu: 100m
            memory: 100Mi
        env:
        - name: GET_HOSTS_FROM
          value: dns
          # If your cluster config does not include a dns service, then to
          # instead access environment variables to find service host
          # info, comment out the 'value: dns' line above, and uncomment the
          # line below.
          # value: env
        ports:
        - containerPort: 80
```

Saving this manifest into `frontend.yaml` and submitting it to a Kubernetes cluster should create the defined ReplicaSet and the pods that it manages.

```
$ kubectl create -f http://k8s.io/examples/controllers/frontend.yaml
replicaset.apps/frontend created
$ kubectl describe rs/frontend
Name:       frontend
Namespace:  default
Selector:   tier=frontend,tier in (frontend)
Labels:     app=guestbook
        tier=frontend
Annotations:    <none>
Replicas:   3 current / 3 desired
Pods Status:    3 Running / 0 Waiting / 0 Succeeded / 0 Failed
Pod Template:
  Labels:       app=guestbook
                tier=frontend
  Containers:
   php-redis:
    Image:      gcr.io/google_samples/gb-frontend:v3
    Port:       80/TCP
    Requests:
      cpu:      100m
      memory:   100Mi
    Environment:
      GET_HOSTS_FROM:   dns
    Mounts:             <none>
  Volumes:              <none>
Events:
  FirstSeen    LastSeen    Count    From                SubobjectPath    Type        Reason
  ---------    --------    -----    ----                -------------    --------    ------
  1m           1m          1        {replicaset-controller }             Normal      Success
  1m           1m          1        {replicaset-controller }             Normal      Success
  1m           1m          1        {replicaset-controller }             Normal      Success
$ kubectl get pods
NAME             READY     STATUS    RESTARTS   AGE
frontend-9si5l   1/1       Running   0          1m
frontend-dnjpy   1/1       Running   0          1m
frontend-qhloh   1/1       Running   0          1m
```

Writing a ReplicaSet Spec

As with all other Kubernetes API objects, a ReplicaSet needs the `apiVersion`, `kind`, and `metadata` fields. For general information about working with manifests, see object management using kubectl.

A ReplicaSet 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 that it is nested and does not have an `apiVersion` or `kind`.

In addition to required fields of a pod, a pod template in a ReplicaSet must specify appropriate labels and an appropriate restart policy.

For labels, make sure to not overlap with other controllers. For more information, see pod selector.

For restart policy, the only allowed value for `.spec.template.spec.restartPolicy` is `Always`, which is the default.

For local container restarts, ReplicaSet delegates to an agent on the node, for example the Kubelet or Docker.

Pod Selector

The `.spec.selector` field is a label selector. A ReplicaSet 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 ReplicaSet to be replaced without affecting the running pods.

The `.spec.template.metadata.labels` must match the `.spec.selector`, or it will be rejected by the API.

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.

Also you should not normally create any pods whose labels match this selector, either directly, with another ReplicaSet, or with another controller such as a Deployment. If you do so, the ReplicaSet 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.

Labels on a ReplicaSet

The ReplicaSet can itself have labels (`.metadata.labels`). Typically, you would set these the same as the `.spec.template.metadata.labels`. However, they are allowed to be different, and the `.metadata.labels` do not affect the behavior of the ReplicaSet.

Replicas

You can specify how many pods should run concurrently by setting `.spec.replicas`. 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 shut down, and a replacement starts early.

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 delete option. e.g. :

```
kubectl proxy --port=8080
curl -X DELETE  'localhost:8080/apis/extensions/v1beta1/namespaces/default/replicasets/front
> -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`, e.g. :

```
kubectl proxy --port=8080
curl -X DELETE  'localhost:8080/apis/extensions/v1beta1/namespaces/default/replicasets/front
> -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 rolling update.

Isolating pods from a ReplicaSet

Pods may be removed from a ReplicaSet'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).

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 an 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 create -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
```

Alternatives to ReplicaSet

Deployment (Recommended)

`Deployment` is a higher-level API object that updates its underlying ReplicaSets and their Pods in a similar fashion as `kubectl rolling-update`. Deployments are recommended if you want this rolling update functionality, because unlike `kubectl rolling-update`, they are declarative, server-side, and have additional features. For more information on running a stateless application using a Deployment, please read Run a Stateless Application Using a Deployment.

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.

Edit This Page

ReplicaSet

ReplicaSet is the next-generation Replication Controller. The only difference between a *ReplicaSet* and a *Replication Controller* right now is the selector

support. ReplicaSet supports the new set-based selector requirements as described in the labels user guide whereas a Replication Controller only supports equality-based selector requirements.

- How to use a ReplicaSet
- When to use a ReplicaSet
- Example
- Writing a ReplicaSet Spec
- Working with ReplicaSets
- Alternatives to ReplicaSet

How to use a ReplicaSet

Most `kubectl` commands that support Replication Controllers also support ReplicaSets. One exception is the `rolling-update` command. If you want the rolling update functionality please consider using Deployments instead. Also, the `rolling-update` command is imperative whereas Deployments are declarative, so we recommend using Deployments through the `rollout` command.

While ReplicaSets can be used independently, today it's 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.

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
    matchExpressions:
      - {key: tier, operator: In, values: [frontend]}
  template:
    metadata:
      labels:
        app: guestbook
        tier: frontend
    spec:
      containers:
      - name: php-redis
        image: gcr.io/google_samples/gb-frontend:v3
        resources:
          requests:
            cpu: 100m
            memory: 100Mi
        env:
        - name: GET_HOSTS_FROM
          value: dns
          # If your cluster config does not include a dns service, then to
          # instead access environment variables to find service host
          # info, comment out the 'value: dns' line above, and uncomment the
          # line below.
          # value: env
        ports:
        - containerPort: 80
```

Saving this manifest into `frontend.yaml` and submitting it to a Kubernetes cluster should create the defined ReplicaSet and the pods that it manages.

```
$ kubectl create -f http://k8s.io/examples/controllers/frontend.yaml
replicaset.apps/frontend created
$ kubectl describe rs/frontend
Name:       frontend
Namespace:  default
Selector:   tier=frontend,tier in (frontend)
Labels:     app=guestbook
        tier=frontend
Annotations:    <none>
Replicas:   3 current / 3 desired
Pods Status:    3 Running / 0 Waiting / 0 Succeeded / 0 Failed
Pod Template:
  Labels:       app=guestbook
                tier=frontend
  Containers:
   php-redis:
    Image:      gcr.io/google_samples/gb-frontend:v3
    Port:       80/TCP
    Requests:
      cpu:      100m
      memory:   100Mi
    Environment:
      GET_HOSTS_FROM:   dns
    Mounts:             <none>
  Volumes:              <none>
Events:
  FirstSeen    LastSeen    Count    From                SubobjectPath    Type        Reason
  ---------    --------    -----    ----                -------------    --------    ------
  1m           1m          1        {replicaset-controller }             Normal      Success
  1m           1m          1        {replicaset-controller }             Normal      Success
  1m           1m          1        {replicaset-controller }             Normal      Success
$ kubectl get pods
NAME             READY     STATUS    RESTARTS   AGE
frontend-9si5l   1/1       Running   0          1m
frontend-dnjpy   1/1       Running   0          1m
frontend-qhloh   1/1       Running   0          1m
```

Writing a ReplicaSet Spec

As with all other Kubernetes API objects, a ReplicaSet needs the `apiVersion`, `kind`, and `metadata` fields. For general information about working with manifests, see object management using kubectl.

A ReplicaSet 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 that it is nested and does not have an `apiVersion` or `kind`.

In addition to required fields of a pod, a pod template in a ReplicaSet must specify appropriate labels and an appropriate restart policy.

For labels, make sure to not overlap with other controllers. For more information, see pod selector.

For restart policy, the only allowed value for `.spec.template.spec.restartPolicy` is `Always`, which is the default.

For local container restarts, ReplicaSet delegates to an agent on the node, for example the Kubelet or Docker.

Pod Selector

The `.spec.selector` field is a label selector. A ReplicaSet 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 ReplicaSet to be replaced without affecting the running pods.

The `.spec.template.metadata.labels` must match the `.spec.selector`, or it will be rejected by the API.

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.

Also you should not normally create any pods whose labels match this selector, either directly, with another ReplicaSet, or with another controller such as a Deployment. If you do so, the ReplicaSet 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.

Labels on a ReplicaSet

The ReplicaSet can itself have labels (`.metadata.labels`). Typically, you would set these the same as the `.spec.template.metadata.labels`. However, they are allowed to be different, and the `.metadata.labels` do not affect the behavior of the ReplicaSet.

Replicas

You can specify how many pods should run concurrently by setting `.spec.replicas`. 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 shut down, and a replacement starts early.

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 delete option. e.g. :

```
kubectl proxy --port=8080
curl -X DELETE  'localhost:8080/apis/extensions/v1beta1/namespaces/default/replicasets/front
> -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`, e.g. :

```
kubectl proxy --port=8080
curl -X DELETE  'localhost:8080/apis/extensions/v1beta1/namespaces/default/replicasets/front
> -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 rolling update.

Isolating pods from a ReplicaSet

Pods may be removed from a ReplicaSet'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).

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 an 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 create -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
```

Alternatives to ReplicaSet

Deployment (Recommended)

`Deployment` is a higher-level API object that updates its underlying ReplicaSets and their Pods in a similar fashion as `kubectl rolling-update`. Deployments are recommended if you want this rolling update functionality, because unlike `kubectl rolling-update`, they are declarative, server-side, and have additional features. For more information on running a stateless application using a Deployment, please read Run a Stateless Application Using a Deployment.

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.

Edit This Page

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
- Running an example ReplicationController
- Writing a ReplicationController Spec
- Working with ReplicationControllers
- Common usage patterns
- Writing programs for Replication
- Responsibilities of the ReplicationController
- API Object
- Alternatives to ReplicationController
- For more information

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” or “rcs” 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 create -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
  ---------       --------     -----    ----                          -------------    ----
  20s             20s          1        {replication-controller }                      Normal
  20s             20s          1        {replication-controller }                      Normal
  20s             20s          1        {replication-controller }                      Normal
```

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. 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.template.metadata.labels`. However, they are allowed to be different, and the `.metadata.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 `.spec.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.

Rolling update is implemented in the client tool `kubectl rolling-update`. Visit `kubectl rolling-update` task for more concrete examples.

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, rolling-update) 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 in a similar fashion as `kubectl rolling-update`. Deployments are recommended if you want this rolling update functionality, because unlike `kubectl rolling-update`, 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 AP Replication Controller.

Edit This Page

Deployments

A *Deployment* controller provides declarative updates for Pods and ReplicaSets.

You describe a *desired state* in a Deployment object, 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:** You should not manage ReplicaSets owned by a Deployment. All the use cases should be covered by manipulating the Deployment object. Consider opening an issue in the main Kubernetes repository if your use case is not covered below.

- Use Case
- Creating a Deployment
- Updating a Deployment
- Rolling Back a Deployment
- Scaling a Deployment
- Pausing and Resuming a Deployment
- Deployment status
- Clean up Policy

- Use Cases
- Writing a Deployment Spec
- Alternative to Deployments

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:1.15.4
        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 `replicas` field.
- The `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: `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". The requirements are ANDed.

- The `template` field contains the following sub-fields:
  - The Pods are labeled `app: nginx`using the `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 1.15.4.
  - Create one container and name it `nginx` using the `name` field.
  - Run the `nginx` image at version `1.15.4`.
  - Open port `80` so that the container can send and accept traffic.

To create this Deployment, run the following command:

```
kubectl create -f https://k8s.io/examples/controllers/nginx-deployment.yaml
```

> **Note:** You may specify the `--record` flag to write the command executed in the resource annotation `kubernetes.io/change-cause`. It is useful for future introspection, for example to see the commands executed in each Deployment revision.

Next, run `kubectl get deployments`. The output is similar to the following:

```
NAME               DESIRED   CURRENT   UP-TO-DATE   AVAILABLE   AGE
nginx-deployment   3         0         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 cluster.
- `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.
- `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 values in each field correspond to the values in the Deployment specification:

- The number of desired replicas is 3 according to `.spec.replicas` field.
- The number of current replicas is 0 according to the `.status.replicas` field.
- The number of up-to-date replicas is 0 according to the `.status.updatedReplicas` field.
- The number of available replicas is 0 according to the `.status.availableReplicas` field.

To see the Deployment rollout status, run `kubectl rollout status deployment.v1.apps/nginx-deployment`. This command returns the following output:

```
Waiting for rollout to finish: 2 out of 3 new replicas have been updated...
deployment.apps/nginx-deployment successfully rolled out
```

Run the `kubectl get deployments` again a few seconds later:

```
NAME               DESIRED   CURRENT   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 (the Pod status is Ready for at least the value of the Deployment's `.spec.minReadySeconds` field).

To see the ReplicaSet (`rs`) created by the deployment, run `kubectl get rs`:

```
NAME                          DESIRED   CURRENT   READY   AGE
nginx-deployment-2035384211   3         3         3       18s
```

Notice that the name of the ReplicaSet is always formatted as `[DEPLOYMENT-NAME]-[POD-TEMPLATE-HASH-VALU`
The hash value is automatically generated when the Deployment is created.

To see the labels automatically generated for each pod, run `kubectl get pods --show-labels`. The following output is returned:

```
NAME                                READY     STATUS    RESTARTS   AGE       LABELS
nginx-deployment-2035384211-7ci7o   1/1       Running   0          18s       app=nginx,pod-t
nginx-deployment-2035384211-kzszj   1/1       Running   0          18s       app=nginx,pod-t
nginx-deployment-2035384211-qqcnn   1/1       Running   0          18s       app=nginx,pod-t
```

The created ReplicaSet ensures that there are three `nginx` Pods running at all times.

> **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

> **Note:** 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.

Suppose that you now want to update the nginx Pods to use the `nginx:1.9.1` image instead of the `nginx:1.7.9` image.

```
$ kubectl set image deployment.v1.apps/nginx-deployment nginx=nginx:1.9.1 --record
deployment.apps/nginx-deployment image updated
```

Alternatively, you can `edit` the Deployment and change `.spec.template.spec.containers[0].image` from `nginx:1.7.9` to `nginx:1.9.1`:

```
$ kubectl edit deployment.v1.apps/nginx-deployment
deployment.apps/nginx-deployment edited
```

To see the rollout status, run:

```
$ kubectl rollout status deployment.v1.apps/nginx-deployment
Waiting for rollout to finish: 2 out of 3 new replicas have been updated...
deployment.apps/nginx-deployment successfully rolled out
```

After the rollout succeeds, you may want to `get` the Deployment:

```
$ kubectl get deployments
NAME               DESIRED   CURRENT   UP-TO-DATE   AVAILABLE   AGE
nginx-deployment   3         3         3            3           36s
```

The number of up-to-date replicas indicates that the Deployment has updated the replicas to the latest configuration. The current replicas indicates the total replicas this Deployment manages, and the available replicas indicates the number of current replicas that are available.

You can 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
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
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 can ensure that only a certain number of Pods may be down while they are being updated. By default, it ensures that at least 25% less than the desired number of Pods are up (25% max unavailable).

Deployment can also ensure that only a certain number of Pods may be created above the desired number of Pods. By default, it ensures that at most 25% more than 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 number of available Pods is at least 2 and the number of total Pods is at most 4.

```
$ kubectl describe deployments
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:1.9.1
    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-deploy
Normal  ScalingReplicaSet  24s   deployment-controller  Scaled up replica set nginx-deploy
Normal  ScalingReplicaSet  22s   deployment-controller  Scaled down replica set nginx-depl
Normal  ScalingReplicaSet  22s   deployment-controller  Scaled up replica set nginx-deploy
Normal  ScalingReplicaSet  19s   deployment-controller  Scaled down replica set nginx-depl
Normal  ScalingReplicaSet  19s   deployment-controller  Scaled up replica set nginx-deploy
Normal  ScalingReplicaSet  14s   deployment-controller  Scaled down replica set nginx-depl
```

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 object is observed by the Deployment controller, a ReplicaSet is created to bring up the desired Pods if there is no existing ReplicaSet doing so. Existing ReplicaSet controlling Pods whose labels match `.spec.selector` but whose template does not match `.spec.template` are scaled down. Eventually, the new ReplicaSet will be scaled to `.spec.replicas` and all old ReplicaSets will be scaled to 0.

If you update a Deployment while an existing rollout is in progress, the Deployment will create a new ReplicaSet as per the update and start scaling that up, and will roll over the ReplicaSet that it was scaling up previously – it will add it to its list of old ReplicaSets and will start scaling it down.

For example, suppose you create a Deployment to create 5 replicas of `nginx:1.7.9`, but then updates the Deployment to create 5 replicas of `nginx:1.9.1`, when only 3 replicas of `nginx:1.7.9` had been created. In that case, Deployment will immediately start killing the 3 `nginx:1.7.9` Pods that it had created, and will start creating `nginx:1.9.1` Pods. It will not wait for 5 replicas of `nginx:1.7.9` 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 – that is, changing the existing value in a selector key – result in the same behavior as additions.
- Selector removals – that is, removing an existing key from the Deployment selector – do not require any changes in the pod template labels. No existing ReplicaSet is 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:1.91` instead of `nginx:1.9.1`:

```
$ kubectl set image deployment.v1.apps/nginx-deployment nginx=nginx:1.91 --record=true
deployment.apps/nginx-deployment image updated
```

The rollout will be stuck.

```
$ kubectl rollout status deployment.v1.apps/nginx-deployment
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 will 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
NAME                           DESIRED   CURRENT   READY   AGE
nginx-deployment-1564180365   3         3         3       25s
nginx-deployment-2035384211   0         0         0       36s
nginx-deployment-3066724191   1         1         0       6s
```

Looking at the Pods created, you will see that 1 Pod created by new ReplicaSet is stuck in an image pull loop.

```
$ kubectl get pods
NAME                                 READY     STATUS             RESTARTS   AGE
nginx-deployment-1564180365-70iae   1/1       Running            0          25s
nginx-deployment-1564180365-jbqqo   1/1       Running            0          25s
nginx-deployment-1564180365-hysrc   1/1       Running            0          25s
nginx-deployment-3066724191-08mng   0/1       ImagePullBackOff   0          6s
```

> **Note:** The Deployment controller will stop the bad rollout automatically, and will stop 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%.

```
$ kubectl describe deployment
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:1.91
    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
  --------- --------    -----   ----                    -------------   --------    ------
  1m        1m          1       {deployment-controller }                Normal      ScalingR
  22s       22s         1       {deployment-controller }                Normal      ScalingR
  22s       22s         1       {deployment-controller }                Normal      ScalingR
  22s       22s         1       {deployment-controller }                Normal      ScalingR
  21s       21s         1       {deployment-controller }                Normal      ScalingR
  21s       21s         1       {deployment-controller }                Normal      ScalingR
  13s       13s         1       {deployment-controller }                Normal      ScalingR
  13s       13s         1       {deployment-controller }                Normal      ScalingR
```

To fix this, you need to rollback to a previous revision of Deployment that is stable.

Checking Rollout History of a Deployment

First, check the revisions of this deployment:

```
$ kubectl rollout history deployment.v1.apps/nginx-deployment
deployments "nginx-deployment"
REVISION    CHANGE-CAUSE
1           kubectl create --filename=https://k8s.io/examples/controllers/nginx-deployment.y
2           kubectl set image deployment.v1.apps/nginx-deployment nginx=nginx:1.9.1 --record
3           kubectl set image deployment.v1.apps/nginx-deployment nginx=nginx:1.91 --record=
```

`CHANGE-CAUSE` is copied from the Deployment annotation `kubernetes.io/change-cause` to its revisions upon creation. You could specify the`CHANGE-CAUSE` message by:

- Annotating the Deployment with `kubectl annotate deployment.v1.apps/nginx-deployment kubernetes.io/change-cause="image updated to 1.9.1"`
- Append the `--record` flag to save the `kubectl` command that is making changes to the resource.
- Manually editing the manifest of the resource.

To further see the details of each revision, run:

```
$ kubectl rollout history deployment.v1.apps/nginx-deployment --revision=2
deployments "nginx-deployment" revision 2
  Labels:       app=nginx
          pod-template-hash=1159050644
  Annotations:  kubernetes.io/change-cause=kubectl set image deployment.v1.apps/nginx-deploy
  Containers:
   nginx:
    Image:      nginx:1.9.1
    Port:       80/TCP
     QoS Tier:
        cpu:      BestEffort
```

```
      memory:   BestEffort
    Environment Variables:      <none>
  No volumes.
```

Rolling Back to a Previous Revision

Now you've decided to undo the current rollout and rollback to the previous revision:

```
$ kubectl rollout undo deployment.v1.apps/nginx-deployment
deployment.apps/nginx-deployment
```

Alternatively, you can rollback to a specific revision by specify that in `--to-revision`:

```
$ kubectl rollout undo deployment.v1.apps/nginx-deployment --to-revision=2
deployment.apps/nginx-deployment
```

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.

```
$ kubectl get deployment nginx-deployment
NAME               DESIRED   CURRENT   UP-TO-DATE   AVAILABLE   AGE
nginx-deployment   3         3         3            3           30m

$ kubectl describe deployment nginx-deployment
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/ngin
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:1.9.1
    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-deplo
  Normal  ScalingReplicaSet   11m   deployment-controller  Scaled up replica set nginx-deplo
  Normal  ScalingReplicaSet   11m   deployment-controller  Scaled down replica set nginx-dep
  Normal  ScalingReplicaSet   11m   deployment-controller  Scaled up replica set nginx-deplo
  Normal  ScalingReplicaSet   11m   deployment-controller  Scaled down replica set nginx-dep
  Normal  ScalingReplicaSet   11m   deployment-controller  Scaled up replica set nginx-deplo
  Normal  ScalingReplicaSet   11m   deployment-controller  Scaled down replica set nginx-dep
  Normal  ScalingReplicaSet   11m   deployment-controller  Scaled up replica set nginx-deplo
  Normal  DeploymentRollback  15s   deployment-controller  Rolled back deployment "nginx-dep
  Normal  ScalingReplicaSet   15s   deployment-controller  Scaled down replica set nginx-dep
```

Scaling a Deployment

You can scale a Deployment by using the following command:

```
$ kubectl scale deployment.v1.apps/nginx-deployment --replicas=10
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
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), then the Deployment controller will balance 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.

```
$ kubectl get deploy
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=nginx:sometag
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.

```
$ kubectl get rs
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 will be added to the old ReplicaSet and 2 replicas will be added to the new ReplicaSet. The rollout process should eventually move all replicas to the new ReplicaSet, assuming the new replicas become healthy.

```
$ kubectl get deploy
NAME                DESIRED   CURRENT   UP-TO-DATE   AVAILABLE   AGE
nginx-deployment    15        18        7            8           7m
$ kubectl get rs
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 will allow you to apply multiple fixes in between pausing and resuming without triggering unnecessary rollouts.

For example, with a Deployment that was just created:

```
$ kubectl get deploy
NAME      DESIRED   CURRENT   UP-TO-DATE   AVAILABLE   AGE
nginx     3         3         3            3           1m
$ kubectl get rs
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
deployment.apps/nginx-deployment paused
```

Then update the image of the Deployment:

```
$ kubectl set image deployment.v1.apps/nginx-deployment nginx=nginx:1.9.1
deployment.apps/nginx-deployment image updated
```

Notice that no new rollout started:

```
$ kubectl rollout history deployment.v1.apps/nginx-deployment
deployments "nginx"
REVISION  CHANGE-CAUSE
1   <none>

$ kubectl get rs
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,memor
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
deployment.apps/nginx-deployment resumed
$ kubectl get rs -w
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
^C
$ kubectl get rs
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.v1.apps/nginx-deployment
Waiting for rollout to finish: 2 of 3 updated replicas are available...
deployment.apps/nginx-deployment successfully rolled out
$ 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":6
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 will take 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
<...>
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 might look like 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 quo
      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.v1.apps/nginx-deployment
Waiting for rollout to finish: 2 out of 3 new replicas have been updated...
error: deployment "nginx" exceeded its progress deadline
$ 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.

Use Cases

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.

A Deployment 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 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 an optional 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`.

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. 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.

Rollback To

Field `.spec.rollbackTo` has been deprecated in API versions `extensions/v1beta1` and `apps/v1beta1`, and is no longer supported in API versions starting `apps/v1beta2`. Instead, `kubectl rollout undo` as introduced in Rolling Back to a Previous Revision should be used.

Revision History Limit

A Deployment's revision history is stored in the replica sets it controls.

`.spec.revisionHistoryLimit` is an optional field that specifies the number of old ReplicaSets to retain to allow rollback. Its ideal value depends on the frequency and stability of new Deployments. All old ReplicaSets will be kept by default, consuming resources in `etcd` and crowding the output of `kubectl get rs`, if this field is not set. 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.

More specifically, setting this field to zero means that all old ReplicaSets with 0 replica 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.

Alternative to Deployments

kubectl rolling update

`kubectl rolling update` updates Pods and ReplicationControllers in a similar fashion. But Deployments are recommended, since they are declarative, server side, and have additional features, such as rolling back to any previous revision even after the rolling update is done.

Edit This Page

StatefulSets

StatefulSet is the workload API object used to manage stateful applications.

> **Note:** StatefulSets are stable (GA) in 1.9.

Manages the deployment and scaling of a set of PodsThe smallest and simplest Kubernetes object. A Pod represents a set of running containers on your cluster., *and provides guarantees about the ordering and uniqueness* of these Pods.

Like a DeploymentAn API object that manages a replicated application., 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.

A StatefulSet operates under the same pattern as any other Controller. You define your desired state in a StatefulSet *object*, and the StatefulSet *controller* makes any necessary updates to get there from the current state.

- Using StatefulSets
- Limitations
- Components
- Pod Selector
- Pod Identity
- Deployment and Scaling Guarantees
- Update Strategies
- What's next

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 with a controller that provides a set of stateless replicas. Controllers such as Deployment or ReplicaSet may be better suited to your stateless needs.

Limitations

- StatefulSet was a beta resource prior to 1.9 and not available in any Kubernetes release prior to 1.5.
- 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.

Components

The example below demonstrates the components of a StatefulSet.

- 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.

```
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
```

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.

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}
cluster.local	foo/nginx	foo/web	nginx.foo.svc.cluster.local	web-{0..N-1}
kube.local	foo/nginx	foo/web	nginx.foo.svc.kube.local	web-{0..N-1}

> **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.

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.

What's next

- Follow an example of deploying a stateful application.
- Follow an example of deploying Cassandra with Stateful Sets.

Edit This Page

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, such as `glusterd`, `ceph`, on each node.
- running a logs collection daemon on every node, such as `fluentd` or `logstash`.
- running a node monitoring daemon on every node, such as Prometheus Node Exporter, `collectd`, Dynatrace OneAgent, Datadog agent, New Relic agent, Ganglia `gmond` or Instana agent.

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
- How Daemon Pods are Scheduled
- Communicating with Daemon Pods
- Updating a DaemonSet
- Alternatives to DaemonSet

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:
      - key: node-role.kubernetes.io/master
        effect: NoSchedule
      containers:
      - name: fluentd-elasticsearch
        image: k8s.gcr.io/fluentd-elasticsearch:1.20
        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
```

`controllers/daemonset.yaml`

- Create a DaemonSet based on the YAML file: `kubectl create -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 deploying applications, configuring containers, and object management using kubectl documents.

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.

Also you should not normally create any Pods whose labels match this selector, either directly, via another DaemonSet, or via other controller such as ReplicaSet. Otherwise, the DaemonSet controller will think that those Pods were created by it. Kubernetes will not stop you from doing this. One case where you might want to do this is manually create a Pod with a different value on a node for testing.

Running Pods on Only Some Nodes

If you specify a `.spec.template.spec.nodeSelector`, 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 DaemonSet controller (disabled by default since 1.12)

Normally, the machine that a Pod runs on is selected by the Kubernetes scheduler. However, Pods created by the DaemonSet controller have the machine already selected (`.spec.nodeName` is specified when the Pod is created, so it is ignored by the scheduler). Therefore:

- The `unschedulable` field of a node is not respected by the DaemonSet controller.
- The DaemonSet controller can make Pods even when the scheduler has not been started, which can help cluster bootstrap.

Scheduled by default scheduler (enabled by default since 1.12)

FEATURE STATE: `Kubernetes v1.12` beta

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

- The version names contain beta (e.g. v2beta3).
- Code is well tested. Enabling the feature is considered safe. Enabled by default.
- Support for the overall feature will not be dropped, though details may change.

- The schema and/or semantics of objects may change in incompatible ways in a subsequent beta or stable release. When this happens, we will provide instructions for migrating to the next version. This may require deleting, editing, and re-creating API objects. The editing process may require some thought. This may require downtime for applications that rely on the feature.
- Recommended for only non-business-critical uses because of potential for incompatible changes in subsequent releases. If you have multiple clusters that can be upgraded independently, you may be able to relax this restriction.
- **Please do try our beta features and give feedback on them! After they exit beta, it may not be practical for us to make more changes.**

A 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 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	Alpha Features	Version	Description
`node.kubernetes.io/not-ready`	NoExecute	`TaintBasedEvictions`	1.8+	When `Tain`
`node.kubernetes.io/unreachable`	NoExecute	`TaintBasedEvictions`	1.8+	When `Tain`
`node.kubernetes.io/disk-pressure`	NoSchedule		1.8+	
`node.kubernetes.io/memory-pressure`	NoSchedule		1.8+	
`node.kubernetes.io/unschedulable`	NoSchedule		1.12+	DaemonSet
`node.kubernetes.io/network-unavailable`	NoSchedule		1.12+	DaemonSet

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. You can then create a new DaemonSet with a different template. The new DaemonSet with the different template will recognize all the existing Pods as having matching labels. It will not modify or delete them despite a mismatch in the Pod template. You will need to force new Pod creation by deleting the Pod or deleting the node.

In Kubernetes version 1.6 and later, 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.

Edit This Page

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
- Controlling how the garbage collector deletes dependents
- Known issues
- What's next

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 create -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
  ...
```

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` 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".

Prior to Kubernetes 1.9, the default garbage collection policy for many controller resources was `orphan`. This included ReplicationController, ReplicaSet, StatefulSet, DaemonSet, and Deployment. For kinds in the `extensions/v1beta1`, `apps/v1beta1`, and `apps/v1beta2` group versions, unless you specify otherwise, dependent objects are orphaned by default. In Kubernetes 1.9, for all kinds in the `apps/v1` group version, dependent objects are deleted by default.

Here's an example that deletes dependents in background:

```
kubectl proxy --port=8080
curl -X DELETE localhost:8080/apis/apps/v1/namespaces/default/replicasets/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

Edit This Page

TTL Controller for Finished Resources

FEATURE STATE: `Kubernetes v1.12` alpha

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

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

The TTL controller provides a TTL 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 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, i.e. 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

Edit This Page

Jobs - Run to Completion

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 job itself is complete. Deleting a Job will cleanup 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).

A Job can also be used to run multiple pods in parallel.

- Running an example Job
- Writing a Job Spec
- Handling Pod and Container Failures
- Job Termination and Cleanup
- Clean Up Finished Jobs Automatically
- Job Patterns
- Advanced Usage
- Alternatives
- Cron Jobs

Running an example Job

Here is an example Job config. It computes 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
```

Run the example job by downloading the example file and then running this command:

```
$ kubectl create -f https://k8s.io/examples/controllers/job.yaml
job "pi" created
```

Check on the status of the job using this command:

```
$ kubectl describe jobs/pi
Name:             pi
Namespace:        default
Selector:         controller-uid=b1db589a-2c8d-11e6-b324-0209dc45a495
Labels:           controller-uid=b1db589a-2c8d-11e6-b324-0209dc45a495
                  job-name=pi
Annotations:      <none>
Parallelism:      1
Completions:      1
Start Time:       Tue, 07 Jun 2016 10:56:16 +0200
Pods Statuses:    0 Running / 1 Succeeded / 0 Failed
Pod Template:
  Labels:       controller-uid=b1db589a-2c8d-11e6-b324-0209dc45a495
                job-name=pi
  Containers:
   pi:
    Image:      perl
    Port:
    Command:
      perl
      -Mbignum=bpi
      -wle
      print bpi(2000)
    Environment:        <none>
    Mounts:             <none>
  Volumes:              <none>
Events:
  FirstSeen    LastSeen    Count    From            SubobjectPath    Type        Reason
  ---------    --------    -----    ----            -------------    --------    ------
  1m           1m          1        {job-controller }                Normal      Successful
```

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-aiw0a
```

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
3.14159265358979323846264338327950288419716939937510582097494459230781640628620899862803482
```

Writing a Job Spec

As with all other Kubernetes config, a Job needs `apiVersion`, `kind`, and `metadata` fields.

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 Jobs

There are three main types of jobs:

1. Non-parallel Jobs
   - normally only one pod is started, unless the pod fails.
   - job is complete as soon as Pod terminates successfully.
2. Parallel Jobs with a *fixed completion count*:
   - specify a non-zero positive value for `.spec.completions`.
   - the job is complete when there is one successful pod for each value in the range 1 to `.spec.completions`.
   - **not implemented yet:** Each pod 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 with themselves or an external service to determine what each should work on.
    - each pod is independently capable of determining whether or not all its peers are done, thus the entire Job is done.
    - when *any* pod 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 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 controller has not had time to react.
- If the controller failed to create pods for any reason (lack of ResourceQuota, lack of permission, etc.), then there may be fewer pods than requested.
- The controller may throttle new pod creation due to excessive previous pod failures in the same Job.
- When a pod is gracefully shutdown, 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 pods-states 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. Therefore, your program 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 if no new failed Pods appear before the Job's next status check.

> **Note:** Issue #54870 still exists for versions of Kubernetes prior to version 1.12

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, at which point the Job defers to the `.spec.backoffLimit` described above. 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`, the Job and all of its Pods are terminated. The result is that the job has a status 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.

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

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

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

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 unmo
Job Template Expansion			
Queue with Pod Per Work Item			sometime
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 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	.spec.completions	.spec.parallelism
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 of the 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:

```
kind: Job
metadata:
  name: old
  ...
spec:
  selector:
    matchLabels:
      job-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 `job-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:
      job-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 jobs).

As discussed in Pod Lifecycle, `Job` is *only* appropriate for pods with `RestartPolicy` 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 complete control over what pods are created and how work is assigned to them.

Cron Jobs

Support for creating Jobs at specified times/dates (i.e. cron) is available in Kubernetes 1.4. More information is available in the cron job documents

Edit This Page

CronJob

A *Cron Job* creates Jobs on a time-based 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.

> **Note:** All **CronJob** `schedule:` times are denoted in UTC.

For instructions on creating and working with cron jobs, and for an example of a spec file for a cron job, see Running automated tasks with cron jobs.

- Cron Job Limitations

Cron Job 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 decr`

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 cron job is set to start at exactly `08:30:00` and its `startingDeadlineSeconds` is set to 10, if the CronJob controller happens to be down from `08:29:00` to `08:42:00`, the job will not start. Set a longer `startingDeadlineSeconds` if starting later is better than not starting at all.

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.

Edit This Page

Scheduler Performance Tuning

FEATURE STATE: `Kubernetes 1.12` alpha

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

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

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 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”.

- Percentage of Nodes to Score

Percentage of Nodes to Score

Before Kubernetes 1.12, Kube-scheduler used to check the feasibility of all the nodes in a cluster and then scored the feasible ones. Kubernetes 1.12 has a new feature that allows the scheduler to stop looking for more feasible nodes once it finds a certain number of them. This improves the scheduler's performance in large clusters. The number is specified as a percentage of the cluster size and is controlled by a configuration option called `percentageOfNodesToScore`. The range should be between 1 and 100. Other values are considered as 100%. The default value of this option is 50%. A cluster administrator can change this value by providing a different value in the scheduler configuration. However, it may not be necessary to change this value.

```
apiVersion: componentconfig/v1alpha1
kind: KubeSchedulerConfiguration
algorithmSource:
  provider: DefaultProvider

...

percentageOfNodesToScore: 50
```

> **Note:** In clusters with zero or 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.

To disable this feature, you can set `percentageOfNodesToScore` to 100.

Tuning percentageOfNodesToScore

`percentageOfNodesToScore` must be a value between 1 and 100 with the default value of 50. There is also a hardcoded minimum value of 50 nodes which is applied internally. The scheduler tries to find at least 50 nodes regardless of the value of `percentageOfNodesToScore`. This means that changing this option to lower values in clusters with several hundred nodes will not have much impact on the number of feasible nodes that the scheduler tries to find. This is intentional as this option is unlikely to improve performance noticeably in smaller clusters. In large clusters with over a 1000 nodes setting this value to lower numbers may show a noticeable performance improvement.

An important note 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. For this reason, the value should not be set to very low percentages. A general rule of thumb is to never set the value to anything lower than 30. Lower values should

be used only when 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.

It is not recommended to lower this value from its default if your cluster has only several hundred Nodes. It is unlikely to improve the scheduler's performance significantly.

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.

Edit This Page

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
- "Naked" Pods vs ReplicaSets, Deployments, and Jobs
- Services

- Using Labels
- Container Images
- Using kubectl

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 create` 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 vs 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>
```

If you are writing code that talks to a Service, don't use these environment variables; use the DNS name of the Service instead. Service environment variables are provided only for older software which can't be modified to use DNS lookups, and are a much less flexible way of accessing Services.

- 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 `protocol` 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`: the image is pulled every time the pod is started.
- `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, for example `sha256:45b23dee08af5e43a7fea6c4cf9c25ccf269ee113168c19722f87876677c5cb2`. 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>` or `kubectl create -f <directory>`. This looks for Kubernetes configuration in all `.yaml`, `.yml`, and `.json` files in `<directory>` and passes it to `apply` or `create`.
- 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 run` 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.

Edit This Page

Managing Compute Resources for Containers

When you specify a Pod, you can optionally specify how much CPU and memory (RAM) each Container needs. When Containers have resource requests specified, the scheduler can make better decisions about which nodes to place Pods on. And when Containers have their limits specified, contention for resources on a node can be handled in a specified manner. For more details about the difference between requests and limits, see Resource QoS.

- Resource types
- Resource requests and limits of Pod and Container
- Meaning of CPU
- Meaning of memory
- How Pods with resource requests are scheduled
- How Pods with resource limits are run
- Monitoring compute resource usage
- Troubleshooting
- Local ephemeral storage
- Extended resources
- Planned Improvements
- What's next

Resource types

CPU and *memory* are each a *resource type*. A resource type has a base unit. CPU is specified in units of cores, and memory is specified in units of bytes.

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.requests.cpu`
- `spec.containers[].resources.requests.memory`

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.

Meaning of CPU

Limits and requests for CPU resources are measured in *cpu* units. One cpu, in Kubernetes, is equivalent to:

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

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 integer 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 (2^{26} 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: db
    image: mysql
    env:
    - name: MYSQL_ROOT_PASSWORD
      value: "password"
    resources:
      requests:
        memory: "64Mi"
        cpu: "250m"
      limits:
        memory: "128Mi"
        cpu: "500m"
  - name: wp
    image: wordpress
    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 resource usage

The resource usage of a Pod is reported as part of the Pod status.

If optional monitoring is configured for your cluster, then Pod resource usage can be retrieved from the monitoring system.

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 PodExcee
```

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-minion-group-4lw4
Name:            e2e-test-minion-group-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 Reques
  ---------    ----                                  ------------  ----------  -------------
  kube-system  fluentd-gcp-v1.38-28bv1               100m (5%)     0 (0%)      200Mi (2%)
  kube-system  kube-dns-3297075139-61lj3             260m (13%)    0 (0%)      100Mi (1%)
  kube-system  kube-proxy-e2e-test-...               100m (5%)     0 (0%)      0 (0%)
  kube-system  monitoring-influxdb-grafana-v4-z1m12  200m (10%)    200m (10%)  600Mi (8%)
  kube-system  node-problem-detector-v0.1-fj7m3      20m (1%)      200m (10%)  20Mi (0%)
Allocated resources:
  (Total limits may be over 100 percent, i.e., overcommitted.)
  CPU Requests    CPU Limits    Memory Requests    Memory Limits
  ------------    ----------    ---------------    -------------
  680m (34%)      400m (20%)    920Mi (12%)        1070Mi (14%)
```

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:

```
[12:54:41] $ 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:                      Running
      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
  Tue, 07 Jul 2015 12:53:51 -0700   Tue, 07 Jul 2015 12:53:51 -0700  1      {scheduler }
  Tue, 07 Jul 2015 12:53:51 -0700   Tue, 07 Jul 2015 12:53:51 -0700  1      {kubelet kuberne
  Tue, 07 Jul 2015 12:53:51 -0700   Tue, 07 Jul 2015 12:53:51 -0700  1      {kubelet kuberne
```

```
  Tue, 07 Jul 2015 12:53:51 -0700   Tue, 07 Jul 2015 12:53:51 -0700  1      {kubelet kuberne
  Tue, 07 Jul 2015 12:53:51 -0700   Tue, 07 Jul 2015 12:53:51 -0700  1      {kubelet kuberne
```

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:

```
[13:59:01] $ kubectl get pod -o go-template='{{range.status.containerStatuses}}{{"Container
Container Name: simmemleak
LastState: map[terminated:map[exitCode:137 reason:OOM Killed startedAt:2015-07-07T20:58:43Z
```

You can see that the Container was terminated because of `reason:OOM Killed`, where `OOM` stands for Out Of Memory.

Local ephemeral storage

FEATURE STATE: `Kubernetes v1.12` beta

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

- The version names contain beta (e.g. v2beta3).
- Code is well tested. Enabling the feature is considered safe. Enabled by default.
- Support for the overall feature will not be dropped, though details may change.
- The schema and/or semantics of objects may change in incompatible ways in a subsequent beta or stable release. When this happens, we will provide instructions for migrating to the next version. This may require deleting, editing, and re-creating API objects. The editing process may require some thought. This may require downtime for applications that rely on the feature.
- Recommended for only non-business-critical uses because of potential for incompatible changes in subsequent releases. If you have multiple clusters that can be upgraded independently, you may be able to relax this restriction.
- **Please do try our beta features and give feedback on them! After they exit beta, it may not be practical for us to make more changes.**

Kubernetes version 1.8 introduces a new resource, *ephemeral-storage* for managing local ephemeral storage. In each Kubernetes node, kubelet's root directory (/var/lib/kubelet by default) and log directory (/var/log) are stored on the root partition of the node. This partition is also shared and consumed by Pods via emptyDir volumes, container logs, image layers and container writable layers.

This partition is "ephemeral" and applications cannot expect any performance SLAs (Disk IOPS for example) from this partition. Local ephemeral storage

management only applies for the root partition; the optional partition for image layer and writable layer is out of scope.

> **Note:** If an optional runtime partition is used, root partition will not hold any image layer or writable layers.

Requests and limits setting for 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 integer 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
```

For example, the following 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 storage.

```
apiVersion: v1
kind: Pod
metadata:
  name: frontend
spec:
  containers:
  - name: db
    image: mysql
    env:
    - name: MYSQL_ROOT_PASSWORD
      value: "password"
    resources:
      requests:
        ephemeral-storage: "2Gi"
      limits:
        ephemeral-storage: "4Gi"
  - name: wp
    image: wordpress
    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.

How Pods with ephemeral-storage limits run

For container-level isolation, if a Container's writable layer and logs usage exceeds its storage limit, the Pod will be evicted. For pod-level isolation, if the sum of the local ephemeral storage usage from all containers and also the Pod's emptyDir volumes exceeds the limit, the Pod will be evicted.

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
```

Planned Improvements

Kubernetes version 1.5 only allows resource quantities to be specified on a Container. It is planned to improve accounting for resources that are shared by all Containers in a Pod, such as emptyDir volumes.

Kubernetes version 1.5 only supports Container requests and limits for CPU and memory. It is planned to add new resource types, including a node disk space resource, and a framework for adding custom resource types.

Kubernetes supports overcommitment of resources by supporting multiple levels of Quality of Service.

In Kubernetes version 1.5, one unit of CPU means different things on different cloud providers, and on different machine types within the same cloud providers. For example, on AWS, the capacity of a node is reported in ECUs, while in GCE it is reported in logical cores. We plan to revise the definition of the cpu resource to allow for more consistency across providers and platforms.

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.
- Container API
- ResourceRequirements

Edit This Page

Assigning Pods to Nodes

You can constrain a pod to only be able to run on particular nodes or to prefer to run on particular nodes. There are several ways to do this, and they 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, e.g. 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.

You can find all the files for these examples in our docs repo here.

- nodeSelector
- Interlude: built-in node labels
- Affinity and anti-affinity

nodeSelector

`nodeSelector` is the simplest form of 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 turned 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`.

If this fails with an "invalid command" error, you're likely using an older version of kubectl that doesn't have the `label` command. In that case, see the previous version of this guide for instructions on how to manually set labels on a node.

You can verify that it worked by re-running `kubectl get nodes --show-labels` and checking that the node now has a label.

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 create -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. As of Kubernetes v1.4 these labels are

- `kubernetes.io/hostname`
- `failure-domain.beta.kubernetes.io/zone`
- `failure-domain.beta.kubernetes.io/region`
- `beta.kubernetes.io/instance-type`
- `beta.kubernetes.io/os`
- `beta.kubernetes.io/arch`

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.

Affinity and anti-affinity

`nodeSelector` provides a very simple way to constrain pods to nodes with particular labels. The affinity/anti-affinity feature, currently in beta, greatly expands the types of constraints you can express. The key enhancements are

1. the language is more expressive (not just "AND of exact match")
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` (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.

`nodeSelector` continues to work as usual, but will eventually be deprecated, as node affinity can express everything that `nodeSelector` can express.

Node affinity (beta feature)

Node affinity was introduced as alpha in Kubernetes 1.2. 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 availability 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` is satisfied.

If you specify multiple `matchExpressions` associated with `nodeSelectorTerms`, then the pod can be scheduled onto a node **only if all** `matchExpressions` can be 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.

For more information on node affinity, see the design doc.

Inter-pod affinity and anti-affinity (beta feature)

Inter-pod affinity and anti-affinity were introduced in Kubernetes 1.4. 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 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, e.g. 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, i.e. 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: failure-domain.beta.kubernetes.io/zone
    podAntiAffinity:
      preferredDuringSchedulingIgnoredDuringExecution:
      - weight: 100
        podAffinityTerm:
          labelSelector:
            matchExpressions:
            - key: security
              operator: In
              values:
              - S2
          topologyKey: kubernetes.io/hostname
  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 `failure-domain.beta.kubernetes.io/zone` and some value V such that there is at least one node in the cluster with key `failure-domain.beta.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 prefers not to be scheduled onto a node if that node

is already running a pod with label having key "security" and value "S2". (If the `topologyKey` were `failure-domain.beta.kubernetes.io/zone` then it would mean 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 affinity and for `requiredDuringSchedulingIgnoredDuringExecution` pod anti-affinity, empty `topologyKey` is not allowed.
2. 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.
3. For `preferredDuringSchedulingIgnoredDuringExecution` pod anti-affinity, empty `topologyKey` is interpreted as "all topologies" ("all topologies" here is now limited to the combination of `kubernetes.io/hostname`, `failure-domain.beta.kubernetes.io/zone` and `failure-domain.beta.kubernetes.io/region`).
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:1.12-alpine
```

If we create the above two deployments, our three node cluster should look like below.

node-1	node-2	node-3
webserver-1 *cache-1*	*webserver-2* *cache-2*	*webserver-3* *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
NAME                           READY     STATUS    RESTARTS   AGE       IP           NODE
redis-cache-1450370735-6dzlj   1/1       Running   0          8m        10.192.4.2   kube-no
redis-cache-1450370735-j2j96   1/1       Running   0          8m        10.192.2.2   kube-no
redis-cache-1450370735-z73mh   1/1       Running   0          8m        10.192.3.1   kube-no
web-server-1287567482-5d4dz    1/1       Running   0          7m        10.192.2.3   kube-no
web-server-1287567482-6f7v5    1/1       Running   0          7m        10.192.4.3   kube-no
web-server-1287567482-s330j    1/1       Running   0          7m        10.192.3.2   kube-no
```

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.

For more information on inter-pod affinity/anti-affinity, see the design doc.

You may want to check Taints as well, which allow a *node* to *repel* a set of pods.

Edit This Page

Taints and Tolerations

Node affinity, described here, 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.

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. Tolerations are applied to pods, and allow (but do not require) the pods to schedule onto nodes with matching taints.

- Concepts
- Example Use Cases
- Taint based Evictions
- Taint Nodes by Condition

Concepts

You add a taint to a node using kubectl taint. For example,

```
kubectl taint nodes node1 key=value:NoSchedule
```

places a taint on node `node1`. The taint has key `key`, value `value`, and taint effect `NoSchedule`. This means that no pod will be able to schedule onto `node1` unless it has a matching toleration.

To remove the taint added by the command above, you can run:

```
kubectl taint nodes node1 key: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: "key"
  operator: "Equal"
  value: "value"
  effect: "NoSchedule"
```

```
tolerations:
- key: "key"
  operator: "Exists"
  effect: "NoSchedule"
```

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 `value`s are equal

`Operator` defaults to `Equal` if not specified.

> **Note:**
>
> There are two special cases:
>
> - An empty `key` with operator `Exists` matches all keys, values and effects which means this will tolerate everything.
>
> ```
> tolerations:
> - operator: "Exists"
> ```
>
> - An empty `effect` matches all effects with key `key`.
>
> ```
> tolerations:
> - key: "key"
>   operator: "Exists"
> ```

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 any 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 (alpha feature)**: A per-pod-configurable eviction behavior when there are node problems, which is described in the next section.

Taint based Evictions

Earlier we mentioned the `NoExecute` taint effect, which 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

In addition, Kubernetes 1.6 introduced alpha support for representing node problems. In other words, the node controller automatically taints a node when certain condition is 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.

When the `TaintBasedEvictions` alpha feature is enabled (you can do this by including `TaintBasedEvictions=true` in `--feature-gates` for Kubernetes controller manager, such as `--feature-gates=FooBar=true,TaintBasedEvictions=true`), the taints are automatically added by the NodeController (or kubelet) and the normal logic for evicting pods from nodes based on the Ready NodeCondition is disabled.

> **Note:** To maintain the existing rate limiting behavior of pod evictions due to node problems, the system actually adds the taints in a rate-limited way. This prevents massive pod evictions in scenarios such as the master becoming partitioned from the nodes.

This alpha feature, in combination with `tolerationSeconds`, allows a pod to specify how long it should stay bound to a node that has one or both of these problems.

For example, an application with a lot of local state might want to stay bound to node for a long time in the event of network partition, in the hope that the partition will recover and thus the pod eviction can be avoided. The toleration the pod would use in that case would look like

```
tolerations:
- key: "node.alpha.kubernetes.io/unreachable"
  operator: "Exists"
  effect: "NoExecute"
  tolerationSeconds: 6000
```

Note that Kubernetes automatically adds a toleration for `node.kubernetes.io/not-ready` with `tolerationSeconds=300` unless the pod configuration provided by the user already has a toleration for `node.kubernetes.io/not-ready`. Likewise it adds a toleration for `node.alpha.kubernetes.io/unreachable` with `tolerationSeconds=300` unless the pod configuration provided by the user already has a toleration for `node.alpha.kubernetes.io/unreachable`.

These automatically-added tolerations ensure that the default pod behavior of remaining bound for 5 minutes after one of these problems is detected is maintained. The two default tolerations are added by the DefaultTolerationSeconds admission controller.

DaemonSet pods are created with `NoExecute` tolerations for the following taints with no `tolerationSeconds`:

- `node.alpha.kubernetes.io/unreachable`
- `node.kubernetes.io/not-ready`

This ensures that DaemonSet pods are never evicted due to these problems, which matches the behavior when this feature is disabled.

Taint Nodes by Condition

In version 1.12, `TaintNodesByCondition` feature is promoted to beta, so node lifecycle controller automatically creates taints corresponding to Node conditions. 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. Note that `TaintNodesByCondition` only taints nodes with `NoSchedule` effect. `NoExecute` effect is controlled by `TaintBasedEviction` which is an alpha feature and disabled by default.

Starting in Kubernetes 1.8, 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.

Edit This Page

Secrets

Objects of type `secret` are intended to hold sensitive information, such as passwords, OAuth tokens, and ssh keys. Putting this information in a `secret` is safer and more flexible than putting it verbatim in a `pod` definition or in a docker image. See Secrets design document for more information.

- Overview of Secrets
- Details
- Use cases
- Best practices
- Security Properties

Overview of Secrets

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; putting it in a Secret object allows for more control over how it is used, and reduces the risk of accidental exposure.

Users can create secrets, and the system also creates some secrets.

To use a secret, a pod needs to reference the secret. A secret can be used with a pod in two ways: as files in a volume mounted on one or more of its containers, or used by kubelet when pulling images for the pod.

Built-in Secrets

Service Accounts Automatically Create and Attach Secrets with API Credentials

Kubernetes automatically creates secrets which contain credentials for accessing the API and it automatically modifies your pods to use this type of secret.

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 apiserver, this is the recommended workflow.

See the Service Account documentation for more information on how Service Accounts work.

Creating your own Secrets

Creating a Secret Using kubectl create secret

Say that some pods need to access a database. The username and password that the pods should use is in the files `./username.txt` and `./password.txt` on your local machine.

```
# Create files needed for rest of example.
$ echo -n 'admin' > ./username.txt
$ echo -n '1f2d1e2e67df' > ./password.txt
```

The `kubectl create secret` command packages these files into a Secret and creates the object on the Apiserver.

```
$ kubectl create secret generic db-user-pass --from-file=./username.txt --from-file=./passwo
secret "db-user-pass" created
```

You can check that the secret was created like this:

```
$ kubectl get secrets
NAME                  TYPE                                  DATA      AGE
db-user-pass          Opaque                                2         51s

$ kubectl describe secrets/db-user-pass
Name:            db-user-pass
Namespace:       default
Labels:          <none>
Annotations:     <none>

Type:            Opaque

Data
====
password.txt:    12 bytes
username.txt:    5 bytes
```

Note that neither `get` nor `describe` shows the contents of the file by default. This is to protect the secret from being exposed accidentally to someone looking or from being stored in a terminal log.

See decoding a secret for how to see the contents.

Creating a Secret Manually

You can also create a Secret in a file first, in json or yaml format, and then create that object. The Secret contains two maps: data and stringData. The data field is used to store arbitrary data, encoded using base64. The stringData field is provided for convenience, and allows you to provide secret data as unencoded strings.

For example, to store two strings in a Secret using the data field, convert them to base64 as follows:

```
echo -n 'admin' | base64
YWRtaW4=
echo -n '1f2d1e2e67df' | base64
MWYyZDF1MmU2N2Rm
```

Write a Secret that looks like this:

```
apiVersion: v1
kind: Secret
metadata:
  name: mysecret
type: Opaque
data:
  username: YWRtaW4=
  password: MWYyZDF1MmU2N2Rm
```

Now create the Secret using `kubectl create`:

```
$ kubectl create -f ./secret.yaml
secret "mysecret" created
```

For certain scenarios, you may wish to use the stringData field instead. This field allows you to put a non-base64 encoded string directly into the Secret, and the string will be encoded for you when the Secret is created or updated.

A practical example of this might be where you are deploying an application that uses a Secret to store a configuration file, and you want to populate parts of that configuration file during your deployment process.

If your application uses the following configuration file:

```
apiUrl: "https://my.api.com/api/v1"
username: "user"
password: "password"
```

You could store this in a Secret using the following:

```
apiVersion: v1
kind: Secret
metadata:
  name: mysecret
type: Opaque
stringData:
  config.yaml: |-
    apiUrl: "https://my.api.com/api/v1"
    username: {{username}}
    password: {{password}}
```

Your deployment tool could then replace the `{{username}}` and `{{password}}` template variables before running `kubectl create`.

stringData is a write-only convenience field. It is never output when retrieving Secrets. For example, if you run the following command:

```
kubectl get secret mysecret -o yaml
```

The output will be similar to:

```
apiVersion: v1
data:
  config.yaml: YXBpVXJsOiAiaHR0cHM6Ly9teS5hcGkuY29tL2FwaS92MSIKdXNlcm5hbWU6IHt7dXNlcm5hbWV9f
kind: Secret
metadata:
  creationTimestamp: 2018-11-15T20:40:59Z
  name: mysecret
  namespace: default
  resourceVersion: "7225"
  selfLink: /api/v1/namespaces/default/secrets/mysecret
  uid: c280ad2e-e916-11e8-98f2-025000000001
type: Opaque
```

If a field is specified in both data and stringData, the value from stringData is used. For example, the following Secret definition:

```
apiVersion: v1
kind: Secret
metadata:
  name: mysecret
type: Opaque
data:
  username: YWRtaW4=
stringData:
  username: administrator
```

Results in the following secret:

```
apiVersion: v1
data:
  username: YWRtaW5pc3RyYXRvcg==
kind: Secret
metadata:
  creationTimestamp: 2018-11-15T20:46:46Z
  name: mysecret
  namespace: default
  resourceVersion: "7579"
  selfLink: /api/v1/namespaces/default/secrets/mysecret
  uid: 91460ecb-e917-11e8-98f2-025000000001
type: Opaque
```

Where `YWRtaW5pc3RyYXRvcg==` decodes to `administrator`.

The keys of data and stringData must consist of alphanumeric characters, '-', '_' or '.'.

Encoding Note: The serialized JSON and YAML values of secret data are encoded as base64 strings. Newlines are not valid within these strings and must be omitted. When using the `base64` utility on Darwin/macOS users should avoid using the `-b` option to split long lines. Conversely Linux users *should* add the option `-w 0` to `base64` commands or the pipeline `base64 | tr -d '\n'` if `-w` option is not available.

Decoding a Secret

Secrets can be retrieved via the `kubectl get secret` command. For example, to retrieve the secret created in the previous section:

```
$ kubectl get secret mysecret -o yaml
apiVersion: v1
data:
  username: YWRtaW4=
  password: MWYyZDF1MmU2N2Rm
kind: Secret
metadata:
  creationTimestamp: 2016-01-22T18:41:56Z
  name: mysecret
  namespace: default
  resourceVersion: "164619"
  selfLink: /api/v1/namespaces/default/secrets/mysecret
  uid: cfee02d6-c137-11e5-8d73-42010af00002
type: Opaque
```

Decode the password field:

```
$ echo 'MWYyZDF1MmU2N2Rm' | base64 --decode
1f2d1e2e67df
```

Using Secrets

Secrets can be mounted as data volumes or be exposed as environment variables to be used by a container in a pod. They can also be used by other parts of the system, without being directly exposed to the pod. For example, they 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 and/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

We can also control the paths within the volume where Secret keys are projected. You can use `.spec.volumes[].secret.items` field to change 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 also specify the permission mode bits files part of a secret will have. If you don't specify any, `0644` is used by default. You can specify a default mode for the whole 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: 256
```

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.

You can also use mapping, as in the previous example, and specify different permission 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: 511
```

In this case, the file resulting in `/etc/foo/my-group/my-username` will have permission value of `0777`. Owing to JSON limitations, you must specify the mode in decimal notation.

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 base-64 decoded and stored inside these files. This is the result of commands executed inside the container from the example above:

```
$ ls /etc/foo/
username
password
$ cat /etc/foo/username
```

```
admin
$ cat /etc/foo/password
1f2d1e2e67df
```

The program in a container is responsible for reading the secrets from the files.

Mounted Secrets are updated automatically

When a secret being already consumed in a volume is updated, projected keys are eventually updated as well. Kubelet is checking whether the mounted secret is fresh on every periodic sync. However, it is using its local cache for getting the current value of the Secret. The type of the cache is configurable using the (`ConfigMapAndSecretChangeDetectionStrategy` field in KubeletConfiguration struct). It can be either propagated via watch (default), ttl-based, or simply redirecting all requests to directly kube-apiserver. 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 kubelet sync period + cache propagation delay, where cache propagation delay depends on the chosen cache type (it equals to watch propagation delay, ttl of cache, or zero corespondingly).

> **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 base-64 decoded values of the secret data. This is the result of commands executed inside the container from the example above:

```
$ echo $SECRET_USERNAME
admin
$ echo $SECRET_PASSWORD
1f2d1e2e67df
```

Using imagePullSecrets

An imagePullSecret is a way to pass a secret that contains a Docker (or other) image registry password to the Kubelet so it can pull a private image on behalf of your Pod.

Manually specifying an imagePullSecret

Use of imagePullSecrets is described in the images documentation

Arranging for imagePullSecrets to be Automatically Attached

You can manually create an imagePullSecret, and reference it from a serviceAccount. Any pods created with that serviceAccount or that default to use that serviceAccount, will get their imagePullSecret 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 (e.g. 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 API objects reside in a namespace. They can only be referenced by pods in that same namespace.

Individual secrets are limited to 1MB in size. This is to discourage creation of very large secrets which would exhaust apiserver 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.

Kubelet only supports use of secrets for Pods it gets from the API server. This includes any pods created using kubectl, or indirectly via a replication controller. It does not include pods created via the kubelets `--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 via `secretKeyRef` to keys that do not exist in a named Secret will prevent the pod from starting.

Secrets used to populate environment variables via `envFrom` 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
LASTSEEN   FIRSTSEEN   COUNT     NAME            KIND      SUBOBJECT
0s         0s          1         dapi-test-pod   Pod
```

Secret and Pod Lifetime interaction

When a pod is created via the API, there is no check whether 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, 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: 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_r
```

> **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 if they are compromised.

Now we can create a pod which references the secret with the ssh key and consumes it in a volume:

```
kind: Pod
apiVersion: v1
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 prod credentials and another pod which consumes a secret with test environment credentials.

Make the secrets:

```
$ kubectl create secret generic prod-db-secret --from-literal=username=produser --from-liter
secret "prod-db-secret" created
$ kubectl create secret generic test-db-secret --from-literal=username=testuser --from-liter
secret "test-db-secret" created
```

> **Note:**
>
> Special characters such as `$`, `\*`, and `!` require escaping. If the password you are using has special characters, you need to escape them using the `\\` character. For example, if your actual password is `S!B\*d$zDsb`, you should execute the command this way:
>
> ```
> kubectl create secret generic dev-db-secret --from-literal=username=devuser --from-lite
> ```
>
> You do not need to escape special characters in passwords from files (`--from-file`).

Now make the pods:

```
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"
```

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 config template.

You could further simplify the base pod specification by using two Service Accounts: one called, say, `prod-user` with the `prod-db-secret`, and one called, say, `test-user` with the `test-db-secret`. Then, the pod spec can be shortened to, for example:

```
kind: Pod
apiVersion: v1
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 secret volume

In order to make piece of data 'hidden' (i.e., in a file whose name begins with a dot character), simply make that key begin with a dot. For example, when the following secret is mounted into a volume:

```
kind: Secret
apiVersion: v1
metadata:
  name: dotfile-secret
```

```
data:
  .secret-file: dmFsdWUtMg0KDQo=
---
kind: Pod
apiVersion: v1
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 `secret-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 (e.g. 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 secrets API

When deploying applications that interact with the secrets API, access should be limited 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 secrets 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 `secret` objects 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 `secret` objects, 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. It is not written to disk. It is stored in a tmpfs. It is deleted once the pod that depends on it is deleted.

On most Kubernetes-project-maintained distributions, communication between user to the apiserver, and from apiserver to the kubelets, is protected by SSL/TLS. Secrets are protected when transmitted over these channels.

Secret data on nodes is stored in tmpfs volumes and thus does not come to rest on the node.

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.

Risks

- In the API server secret data is stored as plaintext in etcd; therefore:
  - Administrators should limit access to etcd to admin users
  - Secret data in the API server is at rest on the disk that etcd uses; admins may want to wipe/shred disks used by etcd when no longer in use
- 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 apiserver policy does not allow that user to read the secret object, the user could run a pod which exposes the secret.
- If multiple replicas of etcd are run, then the secrets will be shared between them. By default, etcd does not secure peer-to-peer communication with SSL/TLS, though this can be configured.
- Currently, anyone with root on any node can read any secret from the apiserver, 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.

  Note: As of 1.7 encryption of secret data at rest is supported.

Edit This Page

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
- Context
- The KUBECONFIG environment variable
- Merging kubeconfig files
- File references
- What's next

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.

1. Determine the context to use based on the first hit in this chain:
   (a) Use the `--context` command-line flag if it exits.
   (b) Use the `current-context` from the merged kubeconfig files.

An empty context is allowed at this point.

1. 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:
   (a) Use a command-line flag if it exists: `--user` or `--cluster`.
   (b) If the context is non-empty, take the user or cluster from the context.

The user and cluster can be empty at this point.

1. 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:
   (a) Use command line flags if they exist: `--server`, `--certificate-authority`, `--insecure-skip-tls-verify`.
   (b) If any cluster information attributes exist from the merged kubeconfig files, use them.
   (c) If there is no server location, fail.
2. 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:
   (a) Use command line flags if they exist: `--client-certificate`, `--client-key`, `--username`, `--password`, `--token`.
   (b) Use the `user` fields from the merged kubeconfig files.
   (c) If there are two conflicting techniques, fail.
3. 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`

Edit This Page

Pod Priority and Preemption

FEATURE STATE: `Kubernetes 1.11` beta

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

- The version names contain beta (e.g. v2beta3).
- Code is well tested. Enabling the feature is considered safe. Enabled by default.
- Support for the overall feature will not be dropped, though details may change.
- The schema and/or semantics of objects may change in incompatible ways in a subsequent beta or stable release. When this happens, we will provide instructions for migrating to the next version. This may require deleting, editing, and re-creating API objects. The editing process may require some thought. This may require downtime for applications that rely on the feature.
- Recommended for only non-business-critical uses because of potential for incompatible changes in subsequent releases. If you have multiple clusters that can be upgraded independently, you may be able to relax this restriction.
- **Please do try our beta features and give feedback on them! After they exit beta, it may not be practical for us to make more changes.**

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.

In Kubernetes 1.9 and later, Priority also affects scheduling order of Pods and out-of-resource eviction ordering on the Node.

Pod priority and preemption are moved to beta since Kubernetes 1.11 and are enabled by default in this release and later.

In Kubernetes versions where Pod priority and preemption is still an alpha-level feature, you need to explicitly enable it. To use these features in the older versions of Kubernetes, follow the instructions in the documentation for your Kubernetes version, by going to the documentation archive version for your Kubernetes version.

Kubernetes Version	Priority and Preemption State	Enabled by default
1.8	alpha	no
1.9	alpha	no
1.10	alpha	no
1.11	beta	yes

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. To resolve this issue, ResourceQuota is augmented to support Pod priority. An admin can create ResourceQuota for users at specific priority levels, preventing them from creating pods at high priorities. This feature is in beta since Kubernetes 1.12.

- How to use priority and preemption
- How to disable preemption
- PriorityClass
- Pod priority
- Preemption
- Debugging Pod Priority and Preemption
- Interactions of Pod priority and QoS

How to use priority and preemption

To use priority and preemption in Kubernetes 1.11 and later, follow these steps:

1. Add one or more PriorityClasses.
2. 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.

Keep reading for more information about these steps.

If you try the feature and then decide to disable it, you must remove the PodPriority command-line flag or set it to `false`, and then restart the API server and scheduler. After the feature is disabled, the existing Pods keep their priority fields, but preemption is disabled, and priority fields are ignored. If the feature is disabled, you cannot set `priorityClassName` in new Pods.

How to disable preemption

> **Note:** In Kubernetes 1.11, critical pods (except DaemonSet pods, which are still scheduled by the DaemonSet controller) rely on scheduler preemption to be scheduled when a cluster is under resource pressure. For this reason, you will need to run an older version of Rescheduler if you decide to disable preemption. More on this is provided below.

In Kubernetes 1.11 and later, preemption is controlled by a kube-scheduler flag `disablePreemption`, which is set to `false` by default.

This option is available in component configs only and is not available in old-style command line options. Below is a sample component config to disable preemption:

```
apiVersion: componentconfig/v1alpha1
kind: KubeSchedulerConfiguration
algorithmSource:
  provider: DefaultProvider

...

disablePreemption: true
```

Start an older version of Rescheduler in the cluster

When priority or preemption is disabled, we must run Rescheduler v0.3.1 (instead of v0.4.0) to ensure that critical Pods are scheduled when nodes or cluster are under resource pressure. Since critical Pod annotation is still supported in this release, running Rescheduler should be enough and no other changes to the configuration of Pods should be needed.

Rescheduler images can be found at: gcr.io/k8s-image-staging/rescheduler.

In the code, changing the Rescheduler version back to v.0.3.1 is the reverse of this PR.

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 required `value` field. The higher the value, the higher the priority.

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 your existing cluster and enable 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/v1beta1
kind: PriorityClass
metadata:
  name: high-priority
value: 1000000
globalDefault: false
description: "This priority class should be used for XYZ service pods only."
```

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

In Kubernetes 1.9 and later, when Pod priority is enabled, 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 deleted 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, `nominatedNodeName` 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 Pod Disruption Budget (PDB) allows application owners to limit the number Pods of a replicated application that are down simultaneously from voluntary disruptions. Kubernetes 1.9 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: failure-domain.beta.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 we find an algorithm with reasonable performance. We cannot promise anything at this point, and cross Node preemption will not be considered a blocker for Beta or GA.

Debugging Pod Priority and Preemption

Pod Priority and Preemption is a major feature that could potentially disrupt Pod scheduling if it has bugs.

Potential problems caused by Priority and Preemption

The followings are some of the potential problems that could be caused by bugs in the implementation of the feature. This list is not exhaustive.

Pods are preempted unnecessarily

Preemption removes existing Pods from a cluster under resource pressure to make room for higher priority pending Pods. If a user gives high priorities to certain Pods by mistake, these unintentional high priority Pods may cause preemption in the cluster. As mentioned above, Pod priority is specified by setting the `priorityClassName` field of `podSpec`. The integer value of priority is then resolved and populated to the `priority` field of `podSpec`.

To resolve the problem, `priorityClassName` of the Pods must be changed to use lower priority classes or should be left empty. 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 higher 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, but it can be any different value as specified in the PodSpec. If the victim Pods do not terminate within this period they are force-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.

In the absence of such a higher priority Pod, we expect the preemptor Pod to be scheduled after the graceful termination period of the victims is over.

Higher priority Pods are preempted before lower priority pods

The scheduler tries to find nodes that can run a pending Pod and if no node is found, it tries to remove Pods with lower priority from one node 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, we expect the scheduler to choose a node with the lowest priority. If that is not the case, it may indicate a bug in the scheduler.

Interactions of Pod priority and QoS

Pod priority and QoS 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 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 whose 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.

Edit This Page

Scheduler Performance Tuning

FEATURE STATE: `Kubernetes 1.12` alpha

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

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

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 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".

- Percentage of Nodes to Score

Percentage of Nodes to Score

Before Kubernetes 1.12, Kube-scheduler used to check the feasibility of all the nodes in a cluster and then scored the feasible ones. Kubernetes 1.12 has a new feature that allows the scheduler to stop looking for more feasible nodes once it finds a certain number of them. This improves the scheduler's performance in large clusters. The number is specified as a percentage of the cluster size and is controlled by a configuration option called `percentageOfNodesToScore`. The range should be between 1 and 100. Other values are considered as 100%.

The default value of this option is 50%. A cluster administrator can change this value by providing a different value in the scheduler configuration. However, it may not be necessary to change this value.

```
apiVersion: componentconfig/v1alpha1
kind: KubeSchedulerConfiguration
algorithmSource:
  provider: DefaultProvider

...

percentageOfNodesToScore: 50
```

> **Note:** In clusters with zero or 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.

To disable this feature, you can set `percentageOfNodesToScore` to 100.

Tuning percentageOfNodesToScore

`percentageOfNodesToScore` must be a value between 1 and 100 with the default value of 50. There is also a hardcoded minimum value of 50 nodes which is applied internally. The scheduler tries to find at least 50 nodes regardless of the value of `percentageOfNodesToScore`. This means that changing this option to lower values in clusters with several hundred nodes will not have much impact on the number of feasible nodes that the scheduler tries to find. This is intentional as this option is unlikely to improve performance noticeably in smaller clusters. In large clusters with over a 1000 nodes setting this value to lower numbers may show a noticeable performance improvement.

An important note 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. For this reason, the value should not be set to very low percentages. A general rule of thumb is to never set the value to anything lower than 30. Lower values should be used only when 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.

It is not recommended to lower this value from its default if your cluster has only several hundred Nodes. It is unlikely to improve the scheduler's performance significantly.

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.

Edit This Page

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. In 1.7, users can add these custom entries with the HostAliases field in PodSpec.

Modification not using HostAliases is not suggested because the file is managed by Kubelet and can be overwritten on during Pod creation/restart.

- Default Hosts File Content
- Adding Additional Entries with HostAliases
- Why Does Kubelet Manage the Hosts File?

Default Hosts File Content

Lets start an Nginx Pod which is assigned a Pod IP:

```
kubectl run nginx --image nginx --generator=run-pod/v1
```

```
pod/nginx created
```

Examine a Pod IP:

```
kubectl get pods --output=wide
```

```
NAME     READY     STATUS    RESTARTS   AGE    IP           NODE
nginx    1/1       Running   0          13s    10.200.0.4   worker0
```

The hosts file content would look like this:

```
kubectl exec nginx -- cat /etc/hosts
```

```
# Kubernetes-managed hosts file.
127.0.0.1   localhost
::1 localhost ip6-localhost ip6-loopback
fe00::0 ip6-localnet
fe00::0 ip6-mcastprefix
fe00::1 ip6-allnodes
fe00::2 ip6-allrouters
10.200.0.4  nginx
```

By default, the `hosts` file only includes IPv4 and IPv6 boilerplates like `localhost` and its own hostname.

Adding Additional Entries with HostAliases

In addition to the default boilerplate, we can add additional entries to the `hosts` file to resolve `foo.local`, `bar.local` to `127.0.0.1` and `foo.remote`, `bar.remote` to `10.1.2.3`, we can by adding HostAliases to the 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"
```

This Pod can be started with the following commands:

```
kubectl apply -f hostaliases-pod.yaml
```

```
pod/hostaliases-pod created
```

Examine a Pod IP and status:

```
kubectl get pod -o=wide
```

```
NAME                            READY     STATUS      RESTARTS   AGE       IP              NO
hostaliases-pod                 0/1       Completed   0          6s        10.200.0.5      wo
```

The `hosts` file content would look 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
127.0.0.1   bar.local
10.1.2.3    foo.remote
10.1.2.3    bar.remote
```

With the additional entries specified at the bottom.

Why Does Kubelet Manage the Hosts File?

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.

Because of the managed-nature of the file, any user-written content will be overwritten whenever the `hosts` file is remounted by Kubelet in the event of a container restart or a Pod reschedule. Thus, it is not suggested to modify the contents of the file.

Edit This Page

Services

Kubernetes `Pods` are mortal. They are born and when they die, they are not resurrected. `ReplicaSets` in particular create and destroy `Pods` dynamically (e.g. when scaling up or down). While each `Pod` gets its own IP address, even those IP addresses cannot be relied upon to be stable over time. This leads to a problem: if some set of `Pods` (let's call them backends) provides functionality to other `Pods` (let's call them frontends) inside the Kubernetes cluster, how do those frontends find out and keep track of which backends are in that set?

Enter `Services`.

A Kubernetes `Service` is an abstraction which defines a logical set of `Pods` and a policy by which to access them - sometimes called a micro-service. The set of `Pods` targeted by a `Service` is (usually) determined by a `Label Selector` (see below for why you might want a `Service` without a selector).

As an example, consider an 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 or keep track of the list of backends themselves. The `Service` abstraction enables this decoupling.

For Kubernetes-native applications, Kubernetes offers a simple `Endpoints` API that is updated whenever the set of `Pods` in a `Service` changes. For non-native applications, Kubernetes offers a virtual-IP-based bridge to Services which redirects to the backend `Pods`.

- Defining a service
- Virtual IPs and service proxies
- Multi-Port Services
- Choosing your own IP address
- Discovering services
- Headless services
- Publishing services - service types
- Shortcomings
- Future work
- The gory details of virtual IPs
- API Object
- SCTP support
- What's next

Defining a service

A `Service` in Kubernetes is a REST object, similar to a `Pod`. Like all of the REST objects, a `Service` definition can be POSTed to the apiserver to create a new instance. For example, suppose you have a set of `Pods` that each expose port 9376 and carry a label `"app=MyApp"`.

```
kind: Service
apiVersion: v1
metadata:
  name: my-service
spec:
  selector:
    app: MyApp
  ports:
  - protocol: TCP
    port: 80
    targetPort: 9376
```

This specification will create a new `Service` object named "my-service" which targets TCP port 9376 on any `Pod` with the `"app=MyApp"` label. This `Service` will also be assigned an IP address (sometimes called the "cluster IP"), which is used by the service proxies (see below). The `Service`'s selector will be evaluated continuously and the results will be POSTed to an `Endpoints` object also named "my-service".

Note that a `Service` can map an incoming port to any `targetPort`. By default the `targetPort` will be set to the same value as the `port` field. Perhaps more

interesting is that `targetPort` can be a string, referring to the name of a port in the backend `Pods`. The actual port number assigned to that name can be different in each backend `Pod`. This offers a lot of flexibility for deploying and evolving your `Services`. For example, you can change the port number that pods expose in the next version of your backend software, without breaking clients.

Kubernetes `Services` support `TCP`, `UDP` and `SCTP` for protocols. The default is `TCP`.

> **Note:** SCTP support is an alpha feature since Kubernetes 1.12

Services without selectors

Services generally 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 test you use your own databases.
- You want to point your service to a service in another `Namespace` or on another cluster.
- You are migrating your workload to Kubernetes and some of your backends run outside of Kubernetes.

In any of these scenarios you can define a service without a selector:

```
kind: Service
apiVersion: v1
metadata:
  name: my-service
spec:
  ports:
  - protocol: TCP
    port: 80
    targetPort: 9376
```

Because this service has no selector, the corresponding `Endpoints` object will not be created. You can manually map the service to your own specific endpoints:

```
kind: Endpoints
apiVersion: v1
metadata:
  name: my-service
subsets:
  - addresses:
      - ip: 1.2.3.4
    ports:
      - port: 9376
```

> **Note:** The endpoint IPs may not be loopback (127.0.0.0/8), link-local (169.254.0.0/16), or link-local multicast (224.0.0.0/24). They cannot be the cluster IPs of other Kubernetes services either because the `kube-proxy` component doesn't support virtual IPs as destination yet.

Accessing a `Service` without a selector works the same as if it had a selector. The traffic will be routed to endpoints defined by the user (`1.2.3.4:9376` in this example).

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.

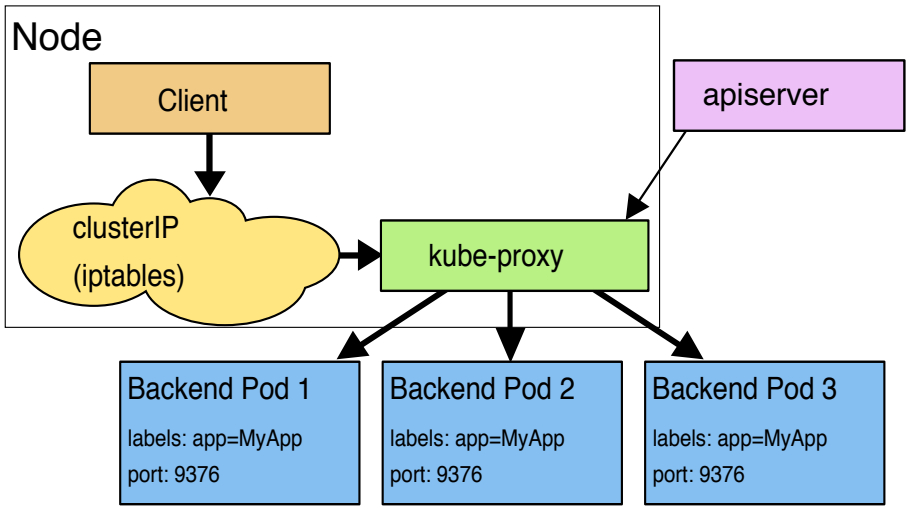
Virtual IPs and service proxies

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`.

In Kubernetes v1.0, `Services` are a "layer 4" (TCP/UDP over IP) construct, the proxy was purely in userspace. In Kubernetes v1.1, the `Ingress` API was added (beta) to represent "layer 7"(HTTP) services, iptables proxy was added too, and became the default operating mode since Kubernetes v1.2. In Kubernetes v1.8.0-beta.0, ipvs proxy was added.

Proxy-mode: userspace

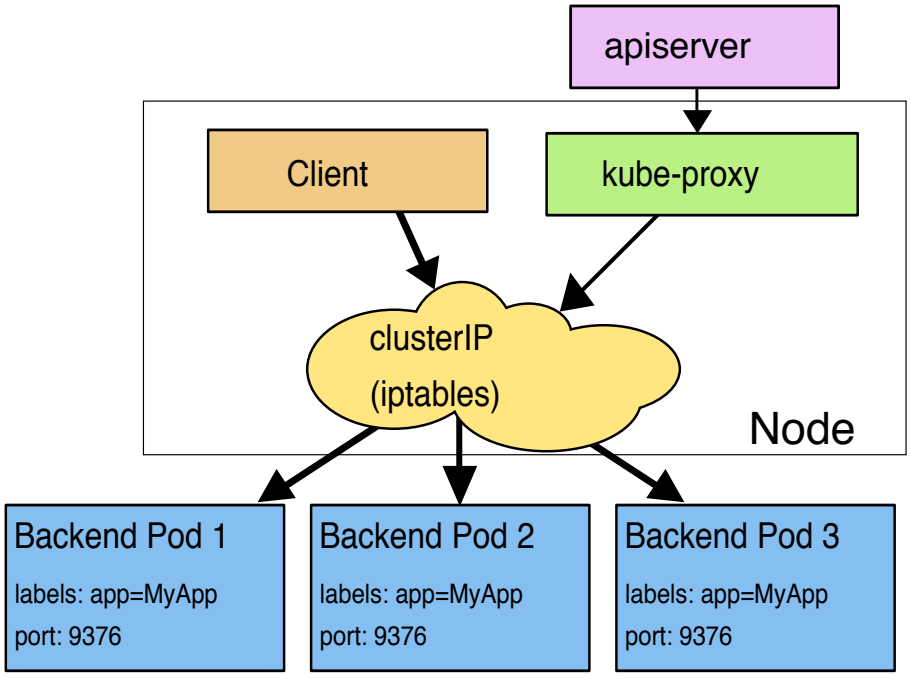
In this mode, kube-proxy watches the Kubernetes master for the addition and removal of `Service` and `Endpoints` objects. For each `Service` it opens a port (randomly chosen) on the local node. Any connections to this "proxy port" will be proxied to one of the `Service`'s backend `Pods` (as reported in `Endpoints`). Which backend `Pod` to use is decided based on the `SessionAffinity` of the `Service`. Lastly, it installs iptables rules which capture traffic to the `Service`'s `clusterIP` (which is virtual) and `Port` and redirects that traffic to the proxy port which proxies the backend `Pod`. By default, the choice of backend is round robin.



Proxy-mode: iptables

In this mode, kube-proxy watches the Kubernetes master for the addition and removal of `Service` and `Endpoints` objects. For each `Service`, it installs iptables rules which capture traffic to the `Service`'s `clusterIP` (which is virtual) and `Port` and redirects that traffic to one of the `Service`'s backend sets. For each `Endpoints` object, it installs iptables rules which select a backend `Pod`. By default, the choice of backend is random.

Obviously, iptables need not switch back between userspace and kernelspace, it should be faster and more reliable than the userspace proxy. However, unlike the userspace proxier, the iptables proxier cannot automatically retry another `Pod` if the one it initially selects does not respond, so it depends on having working readiness probes.



Proxy-mode: ipvs

FEATURE STATE: `Kubernetes v1.9` beta

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

- The version names contain beta (e.g. v2beta3).
- Code is well tested. Enabling the feature is considered safe. Enabled by default.
- Support for the overall feature will not be dropped, though details may change.
- The schema and/or semantics of objects may change in incompatible ways in a subsequent beta or stable release. When this happens, we will provide instructions for migrating to the next version. This may require deleting, editing, and re-creating API objects. The editing process may require some thought. This may require downtime for applications that rely on the feature.
- Recommended for only non-business-critical uses because of potential for incompatible changes in subsequent releases. If you have multiple clusters that can be upgraded independently, you may be able to relax this restriction.
- **Please do try our beta features and give feedback on them! After they exit beta, it may not be practical for us to make more**

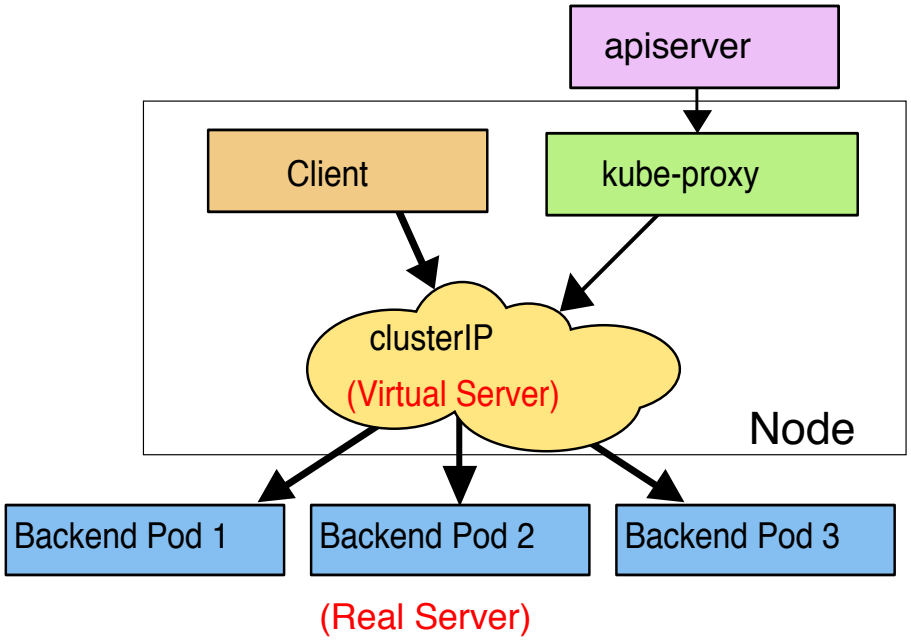
changes.

In this mode, kube-proxy watches Kubernetes Services and Endpoints, calls `netlink` interface to create ipvs rules accordingly and syncs ipvs rules with Kubernetes Services and Endpoints periodically, to make sure ipvs status is consistent with the expectation. When Service is accessed, traffic will be redirected to one of the backend Pods.

Similar to iptables, Ipvs is based on netfilter hook function, but uses hash table as the underlying data structure and works in the kernel space. That means ipvs redirects traffic much faster, and has much better performance when syncing proxy rules. Furthermore, ipvs provides more options for load balancing algorithm, such as:

- `rr`: round-robin
- `lc`: least connection
- `dh`: destination hashing
- `sh`: source hashing
- `sed`: shortest expected delay
- `nq`: never queue

Note: ipvs mode assumes IPVS kernel modules are installed on the node before running kube-proxy. When kube-proxy starts with ipvs proxy mode, kube-proxy would validate if IPVS modules are installed on the node, if it's not installed kube-proxy will fall back to iptables proxy mode.



In any of these proxy model, any 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. Client-IP based session affinity can be selected by setting `service.spec.sessionAffinity` to "ClientIP" (the default is "None"), and you can set the max session sticky time by setting the field `service.spec.sessionAffinityConfig.clientIP.timeoutSeconds` if you have already set `service.spec.sessionAffinity` to "ClientIP" (the default is "10800").

Multi-Port Services

Many `Services` need to expose more than one port. For this case, Kubernetes supports multiple port definitions on a `Service` object. When using multiple ports you must give all of your ports names, so that endpoints can be disambiguated. For example:

```
kind: Service
apiVersion: v1
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 that the port names must only contain lowercase alphanumeric characters and `-`, and must begin & end with an alphanumeric character. `123-abc` and `web` are valid, but `123_abc` and `-web` are not valid names.

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 a user chooses must be a valid IP address and within the `service-cluster-ip-range` CIDR range that is specified by flag to the API server. If the IP address value is invalid, the apiserver returns a 422 HTTP status code to indicate that the value is invalid.

Why not use round-robin DNS?

A question that pops up every now and then is why we do all this stuff with virtual IPs rather than just use standard round-robin DNS. There are a few reasons:

- There is a long history of DNS libraries not respecting DNS TTLs and caching the results of name lookups.
- Many apps do DNS lookups once and cache the results.
- Even if apps and libraries did proper re-resolution, the load of every client re-resolving DNS over and over would be difficult to manage.

We try to discourage users from doing things that hurt themselves. That said, if enough people ask for this, we may implement it as an alternative.

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
```

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.

DNS

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.

For example, if you have a Service called "my-service" in Kubernetes Namespace "my-ns" a DNS record for "my-service.my-ns" is created. Pods which exist in the "my-ns" Namespace should be able to find it by simply doing a name lookup for "my-service". Pods which exist in other Namespaces must qualify the name as "my-service.my-ns". The result of these name lookups is the cluster IP.

Kubernetes also supports DNS SRV (service) records for named ports. If the "my-service.my-ns" Service has a port named "http" with protocol TCP, you can do a DNS SRV query for "_http._tcp.my-service.my-ns" to discover the port number for "http".

The Kubernetes DNS server is the only way to access services of type ExternalName. More information is available in the DNS Pods and Services.

Headless services

Sometimes you don't need or want load-balancing and a single service IP. In this case, you can create "headless" services by specifying `"None"` for the cluster IP (`.spec.clusterIP`).

This option allows developers to reduce coupling to the Kubernetes system by allowing them freedom to do discovery their own way. Applications can still use a self-registration pattern and adapters for other discovery systems could easily be built upon this API.

For such `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 A 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 - service types

For some parts of your application (e.g. frontends) you may want to expose a Service onto an external (outside of your cluster) IP address.

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 will route, 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 will route, 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. This requires version 1.7 or higher of `kube-dns`.

Type NodePort

If you set the `type` field to `NodePort`, the Kubernetes master will allocate a port from a range specified by `--service-node-port-range` flag (default: 30000-32767), and each Node will proxy that port (the same port number on every Node) into your `Service`. That port will be reported in your `Service`'s `.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) (which is supported since Kubernetes v1.10). A comma-delimited list of IP blocks (e.g. 10.0.0.0/8, 1.2.3.4/32) is used to filter addresses local to this node. For example, if you start kube-proxy with flag `--nodeport-addresses=127.0.0.0/8`, kube-proxy will select only the loopback interface for NodePort Services. The `--nodeport-addresses` is defaulted to empty (`[]`), which means select all available interfaces and is in compliance with current NodePort behaviors.

If you want a specific port number, you can specify a value in the `nodePort` field, and the system will allocate you that port or else the API transaction will fail (i.e. you need to take care about possible port collisions yourself). The value you specify must be in the configured range for node ports.

This gives developers the freedom to set up their own load balancers, 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 will be visible as both `<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).)

Type LoadBalancer

On cloud providers which support external load balancers, setting the `type` field to `LoadBalancer` will provision a load balancer for your `Service`. The

actual creation of the load balancer happens asynchronously, and information about the provisioned balancer will be published in the `Service`'s `.status.loadBalancer` field. For example:

```
kind: Service
apiVersion: v1
metadata:
  name: my-service
spec:
  selector:
    app: MyApp
  ports:
  - protocol: TCP
    port: 80
    targetPort: 9376
  clusterIP: 10.0.171.239
  loadBalancerIP: 78.11.24.19
  type: LoadBalancer
status:
  loadBalancer:
    ingress:
    - ip: 146.148.47.155
```

Traffic from the external load balancer will be directed at the backend `Pods`, though exactly how that works depends on the cloud provider. Some cloud providers allow the `loadBalancerIP` to be specified. In those cases, the load-balancer will be created with the user-specified `loadBalancerIP`. If the `loadBalancerIP` field is not specified, an ephemeral IP will be assigned to the loadBalancer. If the `loadBalancerIP` is specified, but the cloud provider does not support the feature, the field will be ignored.

Special notes for Azure: To use user-specified public type `loadBalancerIP`, a static type public IP address resource needs to be created first, and it 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 `CreatingLoadBalancerFailed` permission issues see, Use a static IP address with the Azure Kubernetes Service (AKS) load balancer or CreatingLoadBalancerFailed on AKS cluster with advanced networking.

> **Note:** he support of SCTP in the cloud provider's load balancer is up to the cloud provider's load balancer implementation. If SCTP is not supported by the cloud provider's load balancer the Service creation request is accepted but the creation of the load balancer fails.

Internal load balancer

In a mixed environment it is sometimes necessary to route traffic from services inside the same VPC.

In a split-horizon DNS environment you would need two services to be able to route both external and internal traffic to your endpoints.

This can be achieved by adding the following annotations to the service based on cloud provider.

- Default
- GCP
- AWS
- Azure
- OpenStack

Select one of the tabs.

```
[...]
metadata:
    name: my-service
    annotations:
        cloud.google.com/load-balancer-type: "Internal"
[...]
```

Use `cloud.google.com/load-balancer-type: "internal"` for masters with version 1.7.0 to 1.7.3. For more information, see the docs.

```
[...]
metadata:
    name: my-service
    annotations:
        service.beta.kubernetes.io/aws-load-balancer-internal: 0.0.0.0/0
[...]
```

```
[...]
metadata:
    name: my-service
    annotations:
        service.beta.kubernetes.io/azure-load-balancer-internal: "true"
[...]
```

```
[...]
metadata:
    name: my-service
    annotations:
        service.beta.kubernetes.io/openstack-internal-load-balancer: "true"
[...]
```

SSL support on AWS

For partial SSL support on clusters running on AWS, starting with 1.3 three annotations can be added to a `LoadBalancer` service:

```
metadata:
  name: my-service
  annotations:
    service.beta.kubernetes.io/aws-load-balancer-ssl-cert: arn:aws:acm:us-east-1:12345678901
```

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 will expect the pod to authenticate itself over the encrypted connection.

HTTP and HTTPS will select layer 7 proxying: the ELB will terminate the connection with the user, parse headers and inject the `X-Forwarded-For` header with the user's IP address (pods will only see the IP address of the ELB at the other end of its connection) when forwarding requests.

TCP and SSL will select layer 4 proxying: the ELB will forward traffic without modifying the headers.

In a mixed-use environment where some ports are secured and others are left unencrypted, the following annotations may be used:

```
    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.

Beginning in 1.9, services can use predefined AWS SSL policies for any HTTPS or SSL listeners. To see which policies are available for use, run the awscli command:

```
aws elb describe-load-balancer-policies --query 'PolicyDescriptions[].PolicyName'
```

Any one of those policies can then be specified using the "`service.beta.kubernetes.io/aws-load-balancer-`annotation, for example:

```
metadata:
  name: my-service
  annotations:
    service.beta.kubernetes.io/aws-load-balancer-ssl-negotiation-policy: "ELBSecurityPol
```

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.

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
    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
    # The logical hierarchy you created for your Amazon S3 bucket, for example `my-bucke
```

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-d` 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

    service.beta.kubernetes.io/aws-load-balancer-cross-zone-load-balancing-enabled: "tru
    # Specifies whether cross-zone load balancing is enabled for the load balancer

    service.beta.kubernetes.io/aws-load-balancer-additional-resource-tags: "environment=
    # 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
    # value. Defaults to 5, must be between 2 and 60
```

```
    service.beta.kubernetes.io/aws-load-balancer-extra-security-groups: "sg-53fae93f,sg-
    # A list of additional security groups to be added to ELB
```

Network Load Balancer support on AWS [alpha]

> **Warning:** This is an alpha feature and not recommended for production clusters yet.

Starting in version 1.9.0, Kubernetes supports Network Load Balancer (NLB). 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"
```

Unlike Classic Elastic Load Balancers, Network Load Balancers (NLBs) forward the client's IP through to the node. If a service's `.spec.externalTrafficPolicy` is set to `Cluster`, the client's IP address will not be propagated to the end pods.

By setting `.spec.externalTrafficPolicy` to `Local`, client IP addresses will be 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 pods on the same node.

NLB can also be used 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)
Health Check	TCP	NodePort(s) (`.spec.healthCheckNodePort` for `.spec.externalTrafficPolicy`
Client Traffic	TCP	NodePort(s)
MTU Discovery	ICMP	3,4

Be aware that if `.spec.loadBalancerSourceRanges` is not set, Kubernetes will allow 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.

In order to limit which client IP's can access the Network Load Balancer, specify `loadBalancerSourceRanges`.

```
spec:
  loadBalancerSourceRanges:
  - "143.231.0.0/16"
```

> **Note:** NLB only works with certain instance classes, see the AWS documentation for supported instance types.

Type ExternalName

> **Note:** ExternalName Services are available only with `kube-dns` version 1.7 and later.

Services of type ExternalName map a service to a DNS name (specified using the `spec.externalName` parameter) rather than to a typical selector like `my-service` or `cassandra`. This Service definition, for example, would map the `my-service` Service in the `prod` namespace to `my.database.example.com`:

```
kind: Service
apiVersion: v1
metadata:
  name: my-service
  namespace: prod
spec:
  type: ExternalName
  externalName: my.database.example.com
```

When looking up the host `my-service.prod.svc.CLUSTER`, the cluster DNS service will return 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`.

> **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 `externalIPs`. 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 `ServiceSpec`, `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`)

```
kind: Service
apiVersion: v1
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 will work at small to medium scale, but will not scale to very large clusters with thousands of Services. See the original design proposal for portals for more details.

Using the userspace proxy obscures the source-IP of a packet accessing a `Service`. This makes some kinds of firewalling impossible. The iptables proxier 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 `LoadBalancer` work, but AWS does) but the current API requires it.

Future work

In the future we envision that the proxy policy can become more nuanced than simple round robin balancing, for example master-elected or sharded. We also envision that some `Services` will have "real" load balancers, in which case the VIP will simply transport the packets there.

We intend to improve our support for L7 (HTTP) `Services`.

We intend to have more flexible ingress modes for `Services` which encompass the current `ClusterIP`, `NodePort`, and `LoadBalancer` modes and more.

The gory details of virtual IPs

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 users should not be exposed to situations that could cause their actions to fail through no fault of their own. In this situation, we are looking at network ports - users should not have to choose a port number if that choice might collide with another user. That is an isolation failure.

In order to allow users to choose a port number for their `Services`, we must ensure that no two `Services` can collide. We do 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 IPs, otherwise creations will fail with a message indicating an IP could not be allocated. A background controller is responsible for creating that map (to migrate from older versions of Kubernetes that used in memory locking) as well as checking for invalid assignments due to administrator intervention and cleaning up any IPs that were allocated but which no service currently uses.

IPs and VIPs

Unlike `Pod` IP addresses, which actually route to a fixed destination, `Service` IPs are not actually answered by a single host. Instead, we use `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 VIP and port.

We support three proxy modes - userspace, iptables and ipvs which 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 VIP to this new port, and starts accepting connections on it.

When a client connects to the VIP the iptables rule kicks in, and redirects the packets to the `Service 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 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 installs a series of iptables rules which redirect from the VIP to per-`Service` rules. The per-`Service` rules link to per-`Endpoint` rules which redirect (Destination NAT) to the backends.

When a client connects to the VIP 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 VIP to work, and the client IP is not altered.

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 we 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 conns, locality, weighted, persistence).

API Object

Service is a top-level resource in the Kubernetes REST API. More details about the API object can be found at: Service API object.

SCTP support

FEATURE STATE: `Kubernetes v1.12` alpha

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

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

Kubernetes supports SCTP as a `protocol` value in `Service`, `Endpoint`, `NetworkPolicy` and `Pod` definitions as an alpha feature. To enable this feature, the cluster administrator needs to enable the `SCTPSupport` feature gate on the apiserver, for example, `"--feature-gates=SCTPSupport=true,..."`. When the feature gate is enabled, users can set the `protocol` field of a `Service`, `Endpoint`, `NetworkPolicy` and `Pod` to `SCTP`. Kubernetes sets up the network accordingly for the SCTP associations, just like it does for TCP connections.

Warnings

The support of multihomed SCTP associations

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.

Service with type=LoadBalancer

A `Service` with `type` LoadBalancer and `protocol` SCTP can be created only if the cloud provider's load balancer implementation supports SCTP as a protocol. Otherwise the `Service` creation request is rejected. The current set of cloud load balancer providers (`Azure`, `AWS`, `CloudStack`, `GCE`, `OpenStack`) do not support SCTP.

Windows

SCTP is not supported on Windows based nodes.

Userspace kube-proxy

The kube-proxy does not support the management of SCTP associations when it is in userspace mode.

What's next

Read Connecting a Front End to a Back End Using a Service.

Edit This Page

DNS for Services and Pods

This page provides an overview of DNS support by Kubernetes.

- Introduction
- Services
- Pods
- What's next

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 records

"Normal" (not headless) Services are assigned a DNS A record for a name of the form `my-svc.my-namespace.svc.cluster.local`. This resolves to the cluster IP of the Service.

"Headless" (without a cluster IP) Services are also assigned a DNS A record for a name of the form `my-svc.my-namespace.svc.cluster.local`. 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.local`. For a regular service, this resolves to the port number and the domain name: `my-svc.my-namespace.svc.cluster.local`. 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.local`.

Pods

A Records

When enabled, pods are assigned a DNS A record in the form of "`pod-ip-address.my-namespace.pod.cluster.local`".

For example, a pod with IP `1.2.3.4` in the namespace `default` with a DNS name of `cluster.local` would have an entry: `1-2-3-4.default.pod.cluster.local`.

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.local`".

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
    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
```

```
    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 KubeDNS Server also returns an A 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.local`". DNS serves an A record at that name, pointing to the Pod's IP. Both pods "`busybox1`" and "`busybox2`" can have their distinct A records.

The Endpoints object can specify the `hostname` for any endpoint addresses, along with its IP.

> **Note:** Because A records are not created for Pod names, `hostname` is required for the Pod's A record to be created. A Pod with no `hostname` but with `subdomain` only will only create the A record for the headless service (`default-subdomain.my-namespace.svc.cluster.local`), pointing to the Pod's IP address.

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`": A new option value introduced in Kubernetes v1.9 (Beta in v1.10). 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 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
    command:
      - sleep
      - "3600"
    imagePullPolicy: IfNotPresent
    name: busybox
  restartPolicy: Always
  hostNetwork: true
  dnsPolicy: ClusterFirstWithHostNet
```

Pod's DNS Config

Kubernetes v1.9 introduces an Alpha feature (Beta in v1.10) that allows users more control on the DNS settings for a Pod. This feature is enabled by default in v1.10. To enable this feature in v1.9, the cluster administrator needs to enable the `CustomPodDNS` feature gate on the apiserver and the kubelet, for example, "`--feature-gates=CustomPodDNS=true,...`". When the feature gate is enabled, users can set the `dnsPolicy` field of a Pod to "`None`" and they can add a new field `dnsConfig` to a Pod Spec.

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.local
      - 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.local my.dns.search.suffix
options ndots:2 edns0
```

For IPv6 setup, search path and name server should be setup like this:

```
$ kubectl exec -it busybox -- cat /etc/resolv.conf
nameserver fd00:79:30::a
search default.svc.cluster.local svc.cluster.local cluster.local
options ndots:5
```

What's next

For guidance on administering DNS configurations, check Configure DNS Service

Edit This Page

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 ports across multiple developers 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. We give every pod its own cluster-private-IP address so you do not need to explicitly create links between pods or mapping 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 will elaborate on how you can run reliable services on such a networking model.

This guide uses a simple nginx server to demonstrate proof of concept. The same principles are embodied in a more complete Jenkins CI application.

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 create -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    kubernete
my-nginx-3800858182-kna2y   1/1       Running   0          13s       10.244.2.5    kubernete
```

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 create -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.

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 --replic

$ 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-
my-nginx-3800858182-j4rm4   1/1       Running   0          5s      10.244.3.8    kubernetes-
```

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>        53/UDP,53/TCP   8m
```

If it isn't running, you can enable it. 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 (the CoreDNS cluster addon), so you can talk to the Service from any pod in your cluster using standard methods (e.g. gethostbyname). 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: fa
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 secret KEY=/tmp/nginx.key CERT=/tmp/nginx.crt SECRET=/tmp/secret.json
$ kubectl create -f /tmp/secret.json
secret/nginxsecret created
$ kubectl get secrets
NAME                  TYPE                                  DATA      AGE
default-token-il9rc   kubernetes.io/service-account-token   1         1d
nginxsecret           Opaque                                2         1m
```

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/ngi
#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"
data:
  nginx.crt: "LS0tLS1CRUdJTiBDRVJUSUZJQ0FURS0tLS0tCk1JSURIekNDQWdlZ0F3SUJBZ0lKQUp5M3lQK0pzM
  nginx.key: "LS0tLS1CRUdJTiBQUklWQVRFIEtFWS0tLS0tCk1JSV2UUlCQURBTkJna3Foa2lHOXcwQkFRRUZBQ
```

Now create the secrets using the file:

```
$ kubectl create -f nginxsecrets.yaml
$ kubectl get secrets
NAME                  TYPE                                  DATA      AGE
default-token-il9rc   kubernetes.io/service-account-token   1         1d
nginxsecret           Opaque                                2         1m
```

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
      containers:
      - name: nginxhttps
        image: bprashanth/nginxhttps:1.0
        ports:
        - containerPort: 443
        - containerPort: 80
        volumeMounts:
        - mountPath: /etc/nginx/ssl
          name: secret-volume
```

service/networking/nginx-secure-app.yaml

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 create -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
...
<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   ClusterIP   10.0.162.149   162.222.184.144    80/TCP,81/TCP,82/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.
...
```

What's next

Kubernetes also supports Federated Services, which can span multiple clusters and cloud providers, to provide increased availability, better fault tolerance and greater scalability for your services. See the Federated Services User Guide for further information.

Edit This Page

Ingress

An API object that manages external access to the services in a cluster, typically HTTP.

Ingress can provide load balancing, SSL termination and name-based virtual hosting.

- Terminology
- What is Ingress?
- Prerequisites
- Ingress controllers
- The Ingress Resource
- Before you begin
- Types of Ingress
- Updating an Ingress
- Failing across availability zones
- Future Work

- Alternatives

Terminology

For the sake of clarity, this guide defines the following terms:

- Node: A single virtual or physical machine in a Kubernetes cluster.
- Cluster: A group of nodes firewalled from the internet, that are the primary compute resources managed by Kubernetes.
- 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, added in Kubernetes v1.1, 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.

```
    internet
        |
   [ Ingress ]
   --|-----|--
   [ Services ]
```

An ingress can be configured to give services externally-reachable URLs, load balance traffic, terminate SSL, and offer name based virtual hosting. An ingress controller is responsible for fulfilling the ingress, usually with a loadbalancer, 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

FEATURE STATE: `Kubernetes v1.1` beta

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

- The version names contain beta (e.g. v2beta3).
- Code is well tested. Enabling the feature is considered safe. Enabled by default.
- Support for the overall feature will not be dropped, though details may change.
- The schema and/or semantics of objects may change in incompatible ways in a subsequent beta or stable release. When this happens, we will provide instructions for migrating to the next version. This may require deleting, editing, and re-creating API objects. The editing process may require some thought. This may require downtime for applications that rely on the feature.
- Recommended for only non-business-critical uses because of potential for incompatible changes in subsequent releases. If you have multiple clusters that can be upgraded independently, you may be able to relax this restriction.
- **Please do try our beta features and give feedback on them! After they exit beta, it may not be practical for us to make more changes.**

Before you start using an ingress, there are a few things you should understand. The ingress is a beta resource. You will need an ingress controller to satisfy an ingress, simply creating the resource will have no effect.

GCE/Google Kubernetes Engine deploys an ingress controller on the master. Review the beta limitations of this controller if you are using GCE/GKE.

In environments other than GCE/Google Kubernetes Engine, you may need to deploy an ingress controller. There are a number of ingress controller you may choose from.

Ingress controllers

In order for the ingress resource to work, the cluster must have an ingress controller running. This is unlike other types of controllers, which run as part of the `kube-controller-manager` binary, and are typically started automatically with a cluster. Choose the ingress controller implementation that best fits your cluster.

- Kubernetes as a project currently supports and maintains GCE and nginx controllers.

Additional controllers include:

- Contour is an Envoy based ingress controller provided and supported by Heptio.
- F5 Networks provides support and maintenance for the F5 BIG-IP Controller for Kubernetes.

- HAProxy based ingress controller jcmoraisjr/haproxy-ingress which is mentioned on the blog post HAProxy Ingress Controller for Kubernetes. HAProxy Technologies offers support and maintenance for HAProxy Enterprise and the ingress controller jcmoraisjr/haproxy-ingress.
- Istio based ingress controller Control Ingress Traffic.
- Kong offers community or commercial support and maintenance for the Kong Ingress Controllerfor Kubernetes.
- NGINX, Inc. offers support and maintenance for the NGINX Ingress Controller for Kubernetes.
- Traefik is a fully featured ingress controller (Let's Encrypt, secrets, http2, websocket), and it also comes with commercial support by Containous.

  Note: Review the documentation for your controller to find its specific support policy.

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 provider.

The Ingress Resource

A minimal ingress example:

```
apiVersion: extensions/v1beta1
kind: Ingress
metadata:
  name: test-ingress
  annotations:
    nginx.ingress.kubernetes.io/rewrite-target: /
spec:
  rules:
  - http:
      paths:
      - path: /testpath
        backend:
          serviceName: test
          servicePort: 80
```

Lines 1-6: As with all other Kubernetes resources, an ingress needs `apiVersion`, `kind`, and `metadata` fields.
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 the ingress

controller you are using to learn which annotations are supported and valid options.

Lines 7-9: Ingress spec has all the information needed to configure a loadbalancer or proxy server. Most importantly, it contains a list of rules matched against all incoming requests. Ingress resource only supports rules for directing http traffic.

Lines 10-11: Each http rule contains the following information:

- An optional host. In this example, no host is specified, so the rule will apply to all inbound HTTP traffic through the IP address that will be connected. If a host is provided, for example: foo.bar.com, the rules will apply on to that host)
- a list of paths (e.g.: /testpath) each of which has an associated backend defined with a `serivceName` and `servicePort`. Both the host and path must match the content of an incoming request before the loadbalancer directs traffic to the referenced service.

Lines 12-14: A backend is a service:port combination as described in the services doc. Ingress traffic is typically sent directly to the endpoints matching a backend.

A default backend is often configured in an ingress controller that will service any requests that don't match a path in the spec.

Before you begin

Ideally, all ingress controllers should fulfill this specification, but the various ingress controllers operate slightly differently. The Kubernetes project supports and maintain GCE and nginx ingress controllers.

> **Note:** Make sure you review your ingress controller's specific docs to understand the caveats.

Types of Ingress

Single Service Ingress

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/ingress.yaml

```
apiVersion: extensions/v1beta1
kind: Ingress
metadata:
  name: test-ingress
spec:
  backend:
    serviceName: testsvc
    servicePort: 80
```

If you create it using `kubectl create -f` you should see:

```
kubectl get ingress test-ingress
```

```
NAME           HOSTS     ADDRESS           PORTS     AGE
test-ingress   *         107.178.254.228   80        59s
```

Where `107.178.254.228` 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 will 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 loadbalancers down to a minimum. For example, a setup like:

```
foo.bar.com -> 178.91.123.132 -> / foo    service1:4200
                                 / bar    service2:8080
```

would require an ingress such as:

```
apiVersion: extensions/v1beta1
kind: Ingress
metadata:
  name: simple-fanout-example
  annotations:
    nginx.ingress.kubernetes.io/rewrite-target: /
spec:
  rules:
  - host: foo.bar.com
    http:
```

```
      paths:
      - path: /foo
        backend:
          serviceName: service1
          servicePort: 4200
      - path: /bar
        backend:
          serviceName: service2
          servicePort: 8080
```

When you create the ingress with `kubectl create -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)
Annotations:
  nginx.ingress.kubernetes.io/rewrite-target:  /
Events:
  Type     Reason  Age                From                     Message
  ----     ------  ----               ----                     -------
  Normal   ADD     22s                loadbalancer-controller  default/test
```

The ingress controller will provision an implementation specific loadbalancer that satisfies the ingress, as long as the services (`s1`, `s2`) exist. When it has done so, you will see the address of the loadbalancer 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.

```
foo.bar.com --|                 |-> foo.bar.com s1:80
              | 178.91.123.132  |
bar.foo.com --|                 |-> bar.foo.com s2:80
```

The following ingress tells the backing loadbalancer to route requests based on the Host header.

```
apiVersion: extensions/v1beta1
kind: Ingress
metadata:
  name: name-virtual-host-ingress
spec:
  rules:
  - host: foo.bar.com
    http:
      paths:
      - backend:
          serviceName: service1
          servicePort: 80
  - host: bar.foo.com
    http:
      paths:
      - backend:
          serviceName: service2
          servicePort: 80
```

Default Backends: An ingress with no rules, like the one shown in the previous section, sends all traffic to a single default backend. You can use the same technique to tell a loadbalancer where to find your website's 404 page, by specifying a set of rules *and* a default backend. Traffic is routed to your default backend if none of the Hosts in your ingress match the Host in the request header, and/or none of the paths match the URL of the request.

TLS

You can secure an ingress by specifying a secret that contains a TLS private key and certificate. Currently the ingress only supports a single TLS port, 443, and assumes TLS termination. If the TLS configuration section in an ingress specifies different hosts, they will be 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, e.g.:

```
apiVersion: v1
data:
  tls.crt: base64 encoded cert
  tls.key: base64 encoded key
kind: Secret
metadata:
  name: testsecret-tls
```

```
  namespace: default
type: Opaque
```

Referencing this secret in an ingress will tell the ingress controller to secure the channel from the client to the loadbalancer using TLS. You need to make sure the TLS secret you created came from a certificate that contains a CN for `sslexample.foo.com`.

```
apiVersion: extensions/v1beta1
kind: Ingress
metadata:
  name: tls-example-ingress
spec:
  tls:
  - hosts: sslexample.foo.com
    secretName: testsecret-tls
  rules:
    - host: sslexample.foo.com
      http:
        paths:
        - path: /
          backend:
            serviceName: service1
            servicePort: 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.

Loadbalancing

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 still get these features through the service loadbalancer.

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 which allow you to achieve the same end result. Please review the controller specific docs to see how they handle health checks (nginx, 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:
  Host         Path  Backends
  ----         ----  --------
  foo.bar.com
               /foo   s1:80 (10.8.0.90:80)
Annotations:
  nginx.ingress.kubernetes.io/rewrite-target:  /
Events:
  Type     Reason  Age                From                      Message
  ----     ------  ----               ----                      -------
  Normal   ADD     35s                loadbalancer-controller   default/test
```

```
kubectl edit ingress test
```

This should pop up an editor with the existing yaml, modify it to include the new Host:

```
spec:
  rules:
  - host: foo.bar.com
    http:
      paths:
      - backend:
          serviceName: s1
          servicePort: 80
        path: /foo
  - host: bar.baz.com
    http:
      paths:
      - backend:
          serviceName: s2
          servicePort: 80
        path: /foo
..
```

Saving the yaml will update the resource in the API server, which should tell the ingress controller to reconfigure the loadbalancer.

```
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   s1:80 (10.8.0.90:80)
  bar.baz.com
               /foo   s2:80 (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 by invoking `kubectl replace -f` on a modified ingress yaml file.

Failing across availability zones

Techniques for spreading traffic across failure domains differs between cloud providers. Please check the documentation of the relevant ingress controller for details. You can also refer to the federation documentation for details on deploying ingress in a federated cluster.

Future Work

Track SIG Network for more details on the evolution of the ingress and related resources. You may also track the ingress repository for more details on the evolution of various ingress controllers.

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
- Use a Port Proxy

Edit This Page

Network Policies

A network policy is a specification of how groups of pods are allowed to communicate with each other and other network endpoints.

`NetworkPolicy` resources use labels to select pods and define rules which specify what traffic is allowed to the selected pods.

- Prerequisites
- Isolated and Non-isolated Pods
- The `NetworkPolicy` Resource
- Behavior of `to` and `from` selectors
- Default policies
- SCTP support
- What's next

Prerequisites

Network policies are implemented by the network plugin, so you must be using a networking solution which supports `NetworkPolicy` - simply creating the resource without a controller to implement 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.)

The `NetworkPolicy` Resource

See the NetworkPolicy 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
```

POSTing this to the API server 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 in-

clude 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 whitelist `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 whitelist `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. allows connections to TCP port 6379 of "role=db" pods in the "default" namespace 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. 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.

```
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: default-deny
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.

```
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: allow-all
spec:
  podSelector: {}
  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.

```
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: default-deny
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.

```
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: allow-all
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.

```
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: default-deny
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 v1.12` alpha

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

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

Kubernetes supports SCTP as a `protocol` value in `NetworkPolicy` definitions as an alpha feature. To enable this feature, the cluster administrator needs to enable the `SCTPSupport` feature gate on the apiserver, for example, `"--feature-gates=SCTPSupport=true,..."`. When the feature gate is enabled, users can set the `protocol` field of a `NetworkPolicy` to `SCTP`. Kubernetes sets up the network accordingly for the SCTP associations, just like it does for TCP connections.

The CNI plugin has to support SCTP as `protocol` value in `NetworkPolicy`.

What's next

- See the Declare Network Policy walkthrough for further examples.
- See more Recipes for common scenarios enabled by the NetworkPolicy resource.

Edit This Page

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. In 1.7, users can add these custom entries with the HostAliases field in PodSpec.

Modification not using HostAliases is not suggested because the file is managed by Kubelet and can be overwritten on during Pod creation/restart.

- Default Hosts File Content
- Adding Additional Entries with HostAliases
- Why Does Kubelet Manage the Hosts File?

Default Hosts File Content

Lets start an Nginx Pod which is assigned a Pod IP:

```
kubectl run nginx --image nginx --generator=run-pod/v1
```

```
pod/nginx created
```

Examine a Pod IP:

```
kubectl get pods --output=wide
```

```
NAME     READY     STATUS    RESTARTS   AGE    IP           NODE
nginx    1/1       Running   0          13s    10.200.0.4   worker0
```

The hosts file content would look like this:

```
kubectl exec nginx -- cat /etc/hosts
```

```
# Kubernetes-managed hosts file.
127.0.0.1   localhost
::1 localhost ip6-localhost ip6-loopback
fe00::0 ip6-localnet
fe00::0 ip6-mcastprefix
fe00::1 ip6-allnodes
fe00::2 ip6-allrouters
10.200.0.4  nginx
```

By default, the `hosts` file only includes IPv4 and IPv6 boilerplates like `localhost` and its own hostname.

Adding Additional Entries with HostAliases

In addition to the default boilerplate, we can add additional entries to the `hosts` file to resolve `foo.local`, `bar.local` to `127.0.0.1` and `foo.remote`, `bar.remote` to `10.1.2.3`, we can by adding HostAliases to the 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"
```

This Pod can be started with the following commands:

```
kubectl apply -f hostaliases-pod.yaml
```

```
pod/hostaliases-pod created
```

Examine a Pod IP and status:

```
kubectl get pod -o=wide
```

```
NAME                            READY     STATUS      RESTARTS   AGE       IP              NO
hostaliases-pod                 0/1       Completed   0          6s        10.200.0.5      wo
```

The `hosts` file content would look 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
127.0.0.1   bar.local
10.1.2.3    foo.remote
10.1.2.3    bar.remote
```

With the additional entries specified at the bottom.

Why Does Kubelet Manage the Hosts File?

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.

Because of the managed-nature of the file, any user-written content will be overwritten whenever the `hosts` file is remounted by Kubelet in the event of a container restart or a Pod reschedule. Thus, it is not suggested to modify the contents of the file.

Edit This Page

Volume Snapshots

This document describes the current state of `VolumeSnapshots` in Kubernetes. Familiarity with persistent volumes is suggested.

- Introduction
- Lifecycle of a volume snapshot and volume snapshot content
- Volume Snapshot Contents
- VolumeSnapshots

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.

While `VolumeSnapshots` allow a user to consume abstract storage resources, cluster administrators need to be able to offer a variety of `VolumeSnapshotContents` without exposing users to the details of how those volume snapshots should be provisioned. For these needs there is the `VolumeSnapshotClass` resource.

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, the Kubernetes team provides a sidecar helper container for the snapshot controller called `external-snapshotter`. It watches `VolumeSnapshot` objects and triggers `CreateSnapshot` and `DeleteSnapshot` operations against a CSI endpoint.
- 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 `external-snapshotter`.
- The CSI drivers that support volume snapshot will automatically install CRDs defined for the volume snapshots.

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: statically or dynamically.

Static

A cluster administrator creates a number of `VolumeSnapshotContents`. 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 `VolumeSnapshotContents` the administrator created matches a user's `VolumeSnapshot`, the cluster may try to dynamically provision

a volume snapshot specially for the `VolumeSnapshot` object. This provisioning is based on `VolumeSnapshotClasses`: the `VolumeSnapshot` must request a volume snapshot class and the administrator must have created and configured that class in order for dynamic provisioning to occur.

Binding

A user creates, or has already created in the case of dynamic provisioning, a `VolumeSnapshot` with a specific amount of storage requested and with certain access modes. A control loop watches for new VolumeSnapshots, finds a matching VolumeSnapshotContent (if possible), and binds them together. If a VolumeSnapshotContent was dynamically provisioned for a new VolumeSnapshot, the loop will always bind that VolumeSnapshotContent to the VolumeSnapshot. Once bound, `VolumeSnapshot` binds are exclusive, regardless of how they were bound. A VolumeSnapshot to VolumeSnapshotContent binding is a one-to-one mapping.

VolumeSnapshots will remain unbound indefinitely if a matching VolumeSnapshotContent does not exist. VolumeSnapshots will be bound as matching VolumeSnapshotContents become available.

Delete

Deletion removes both the `VolumeSnapshotContent` object from the Kubernetes API, as well as the associated storage asset in the external infrastructure.

Volume Snapshot Contents

Each VolumeSnapshotContent contains a spec, which is the specification of the volume snapshot.

```
apiVersion: snapshot.storage.k8s.io/v1alpha1
kind: VolumeSnapshotContent
metadata:
  name: new-snapshot-content-test
spec:
  snapshotClassName: csi-hostpath-snapclass
  source:
    name: pvc-test
    kind: PersistentVolumeClaim
  volumeSnapshotSource:
    csiVolumeSnapshotSource:
      creationTime:    1535478900692119403
      driver:          csi-hostpath
```

```
  restoreSize:     10Gi
  snapshotHandle:  7bdd0de3-aaeb-11e8-9aae-0242ac110002
```

Class

A VolumeSnapshotContent can have a class, which is specified by setting the `snapshotClassName` attribute to the name of a VolumeSnapshotClass. A VolumeSnapshotContent of a particular class can only be bound to VolumeSnapshots requesting that class. A VolumeSnapshotContent with no `snapshotClassName` has no class and can only be bound to VolumeSnapshots that request no particular class.

VolumeSnapshots

Each VolumeSnapshot contains a spec and a status, which is the specification and status of the volume snapshot.

```
apiVersion: snapshot.storage.k8s.io/v1alpha1
kind: VolumeSnapshot
metadata:
  name: new-snapshot-test
spec:
  snapshotClassName: csi-hostpath-snapclass
  source:
    name: pvc-test
    kind: PersistentVolumeClaim
```

Class

A volume snapshot can request a particular class by specifying the name of a VolumeSnapshotClass using the attribute `snapshotClassName`. Only VolumeSnapshotContents of the requested class, ones with the same `snapshotClassName` as the VolumeSnapshot, can be bound to the VolumeSnapshot.

Edit This Page

Volumes

On-disk files in a Container are ephemeral, which presents some problems for non-trivial applications when running in Containers. First, when a Container crashes, kubelet will restart it, but the files will be lost - the Container starts with a clean state. Second, when running Containers together in a `Pod` it is

often necessary to share files between those Containers. The Kubernetes `Volume` abstraction solves both of these problems.

Familiarity with Pods is suggested.

- Background
- Types of Volumes
- Using subPath
- Resources
- Out-of-Tree Volume Plugins
- Mount propagation
- What's next

Background

Docker also has a concept of volumes, though it is somewhat looser and less managed. In Docker, a volume is simply a directory on disk or in another Container. Lifetimes are not managed and until very recently there were only local-disk-backed volumes. Docker now provides volume drivers, but the functionality is very limited for now (e.g. as of Docker 1.7 only one volume driver is allowed per Container and there is no way to pass parameters to volumes).

A Kubernetes volume, on the other hand, has an explicit lifetime - the same as the Pod that encloses it. Consequently, a volume outlives any Containers that run within the Pod, and data is preserved across Container restarts. Of course, when a Pod ceases to exist, the volume will cease to exist, too. Perhaps more importantly than this, Kubernetes supports many types of volumes, and a Pod can use any number of them simultaneously.

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, a Pod specifies what volumes to provide for the Pod (the `.spec.volumes` field) and where to mount those into Containers (the `.spec.containers.volumeMounts` field).

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, and any volumes are mounted 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 must independently specify where to mount each volume.

Types of Volumes

Kubernetes supports several types of Volumes:

- awsElasticBlockStore
- azureDisk
- azureFile
- cephfs
- configMap
- csi
- downwardAPI
- emptyDir
- fc (fibre channel)
- flocker
- gcePersistentDisk
- gitRepo (deprecated)
- glusterfs
- hostPath
- iscsi
- local
- nfs
- persistentVolumeClaim
- projected
- portworxVolume
- quobyte
- rbd
- scaleIO
- secret
- storageos
- vsphereVolume

We welcome additional contributions.

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 preserved and the volume is merely unmounted. This means that an EBS volume can be pre-populated with data, and that data can be "handed off" between Pods.

> **Caution:** You must create an EBS volume 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 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. (And also check that the size and EBS volume type are suitable for your use!)

AWS EBS Example configuration

```
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
```

azureDisk

A `azureDisk` is used to mount a Microsoft Azure Data Disk into a Pod.

More details can be found here.

azureFile

A `azureFile` is used to mount a Microsoft Azure File Volume (SMB 2.1 and 3.0) into a Pod.

More details can be found here.

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 "handed off" between Pods. CephFS can be mounted by multiple writers simultaneously.

> **Caution:** You must have your own Ceph server running with the share exported before you can use it.

See the CephFS example for more details.

configMap

The `configMap` resource provides a way to inject configuration data into Pods. The data stored in a `ConfigMap` object can be referenced in a volume of type `configMap` and then consumed by containerized applications running in a Pod.

When referencing a `configMap` object, you can simply provide its name in the volume to reference it. You can also customize the path to use for a specific entry in the ConfigMap. For example, to mount the `log-config` ConfigMap onto a Pod called `configmap-pod`, you might use the YAML below:

```
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`.

> **Caution:** You must create a ConfigMap before you can use it.

> **Note:** A Container using a ConfigMap as a subPath volume mount will not receive ConfigMap updates.

downwardAPI

A `downwardAPI` volume is used to make downward API data available to applications. It mounts a directory and writes the requested data in plain text files.

> **Note:** A Container using Downward API as a subPath volume mount will not receive Downward API updates.

See the `downwardAPI` volume 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, it is initially empty. Containers in the Pod can all 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 forever.

> **Note:** A Container crashing does *NOT* remove a Pod from a node, so 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

By default, `emptyDir` volumes are stored on whatever medium is backing the node - that might be disk or SSD or network storage, depending on your environment. However, you can set the `emptyDir.medium` field to `"Memory"` to tell Kubernetes to mount 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 will count against your Container's memory limit.

Example Pod

```
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 allows an existing fibre channel volume to be mounted in a Pod. You can specify single or multiple target World Wide Names using the parameter `targetWWNs` in your volume configuration. If multiple WWNs are specified, targetWWNs expect that those WWNs are from multi-path connections.

> **Caution:** 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 FC example for more details.

flocker

Flocker is an open-source clustered Container data volume manager. It 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 "handed off" between Pods as required.

> **Caution:** 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 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 "handed off" between Pods.

> **Caution:** 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 PD

A feature of PD is that they can be mounted as read-only by multiple consumers simultaneously. 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 - no simultaneous writers allowed.

Using a PD on a Pod controlled by a ReplicationController will fail unless the PD is read-only or the replica count is 0 or 1.

Creating a PD

Before you can use a GCE PD with a Pod, you need to create it.

```
gcloud compute disks create --size=500GB --zone=us-central1-a my-data-disk
```

Example Pod

```
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

FEATURE STATE: `Kubernetes v1.10` beta

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

- The version names contain beta (e.g. v2beta3).
- Code is well tested. Enabling the feature is considered safe. Enabled by default.
- Support for the overall feature will not be dropped, though details may change.
- The schema and/or semantics of objects may change in incompatible ways in a subsequent beta or stable release. When this happens, we will provide instructions for migrating to the next version. This may require deleting, editing, and re-creating API objects. The editing process may require some thought. This may require downtime for applications that rely on the feature.
- Recommended for only non-business-critical uses because of potential for incompatible changes in subsequent releases. If you have multiple clusters that can be upgraded independently, you may be able to relax this restriction.
- **Please do try our beta features and give feedback on them! After they exit beta, it may not be practical for us to make more changes.**

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 PD:

```
gcloud beta compute disks create --size=500GB my-data-disk
    --region us-central1
    --replica-zones us-central1-a,us-central1-b
```

Example PersistentVolume spec:

```
apiVersion: v1
kind: PersistentVolume
metadata:
  name: test-volume
  labels:
    failure-domain.beta.kubernetes.io/zone: us-central1-a__us-central1-b
spec:
  capacity:
```

```
    storage: 400Gi
  accessModes:
  - ReadWriteOnce
  gcePersistentDisk:
    pdName: my-data-disk
    fsType: ext4
```

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 what can be done as a volume plugin. It mounts an empty directory and clones a git repository into it for your Pod to use. In the future, such volumes may be moved to an even more decoupled model, rather than extending the Kubernetes API for every such use case.

Here is an example for 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 "handed off" between Pods. GlusterFS can be mounted by multiple writers simultaneously.

Caution: 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, user 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
`DirectoryOrCreate`	If nothing exists at the given path, an empty directory will be created there as needed
`Directory`	A directory must exist at the given path
`FileOrCreate`	If nothing exists at the given path, an empty file will be created there as needed with
`File`	A file must exist at the given path
`Socket`	A UNIX socket must exist at the given path
`CharDevice`	A character device must exist at the given path
`BlockDevice`	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
- when Kubernetes adds resource-aware scheduling, as is planned, it will not be able to account for resources used by a `hostPath`
- 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

Example Pod

```
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
```

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 "handed off" between Pods.

> **Caution:** 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 - no simultaneous writers allowed.

See the iSCSI example for more details.

local

FEATURE STATE: `Kubernetes v1.10` beta

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

- The version names contain beta (e.g. v2beta3).

- Code is well tested. Enabling the feature is considered safe. Enabled by default.
- Support for the overall feature will not be dropped, though details may change.
- The schema and/or semantics of objects may change in incompatible ways in a subsequent beta or stable release. When this happens, we will provide instructions for migrating to the next version. This may require deleting, editing, and re-creating API objects. The editing process may require some thought. This may require downtime for applications that rely on the feature.
- Recommended for only non-business-critical uses because of potential for incompatible changes in subsequent releases. If you have multiple clusters that can be upgraded independently, you may be able to relax this restriction.
- **Please do try our beta features and give feedback on them! After they exit beta, it may not be practical for us to make more changes.**

  Note: The alpha PersistentVolume NodeAffinity annotation has been deprecated and will be removed in a future release. Existing PersistentVolumes using this annotation must be updated by the user to use the new PersistentVolume `NodeAffinity` field.

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 yet.

Compared to `hostPath` volumes, local volumes can be used in a durable and portable manner without manually scheduling Pods to nodes, as the system is aware of the volume's node constraints by looking at the node affinity on the PersistentVolume.

However, local volumes are still subject to the availability of the underlying node and are not suitable for all applications. If a node becomes unhealthy, then the local volume will also become inaccessible, and a Pod using it will not be able 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 is an example PersistentVolume spec using a `local` volume and `nodeAffinity`:

```
apiVersion: v1
kind: PersistentVolume
metadata:
  name: example-pv
spec:
```

```
capacity:
  storage: 100Gi
# volumeMode field requires BlockVolume Alpha feature gate to be enabled.
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
```

PersistentVolume `nodeAffinity` is required when using local volumes. It enables the Kubernetes scheduler to correctly schedule Pods using local volumes to the correct node.

PersistentVolume `volumeMode` can now be set to "Block" (instead of the default value "Filesystem") to expose the local volume as a raw block device. The `volumeMode` field requires `BlockVolume` Alpha feature gate to be enabled.

When using local volumes, it is recommended to create a StorageClass with `volumeBindingMode` set to `WaitForFirstConsumer`. See the 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 your 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 "handed off" between Pods. NFS can be mounted by multiple writers simultaneously.

> **Caution:** 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. PersistentVolumes 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 PersistentVolumes example for more details.

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.

The projection of service account tokens is a feature introduced in Kubernetes 1.11 and promoted to Beta in 1.12. To enable this feature on 1.11, you need to explicitly set the `TokenRequestProjection` feature gate to True.

Example Pod with a secret, a downward API, 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 Pod with multiple 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. Below is an 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 Pod containers to access the Kubernetes API server, for example. 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.

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 Kubernetes Pod. Here is an example Pod referencing a pre-provisioned PortworxVolume:

```
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>"
```

> **Caution:** Make sure you have an existing PortworxVolume with name `pxvol` before using it in the Pod.

More details and examples can be found here.

quobyte

A `quobyte` volume allows an existing Quobyte volume to be mounted into your Pod.

> **Caution:** You must have your own Quobyte setup running with the volumes created before you can use it.

See the Quobyte example for more details.

rbd

An `rbd` volume allows a Rados Block Device volume to be mounted into your Pod. Unlike `emptyDir`, which is erased when a Pod is removed, the contents of a `rbd` volume are preserved and the volume is merely unmounted. This means that a RBD volume can be pre-populated with data, and that data can be "handed off" between Pods.

> **Caution:** You must have your own 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 - no simultaneous writers allowed.

See the RBD example for more details.

scaleIO

ScaleIO is a software-based storage platform that can use existing hardware to create clusters of scalable shared block networked storage. The `scaleIO` volume plugin allows deployed Pods to access existing ScaleIO volumes (or it can dynamically provision new volumes for persistent volume claims, see ScaleIO Persistent Volumes).

> **Caution:** You must have an existing ScaleIO cluster already setup and running with the volumes created before you can use them.

The following is an example 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: sp1
      volumeName: vol-0
      secretRef:
        name: sio-secret
      fsType: xfs
```

For further detail, please the 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.

> **Caution:** 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.

Secrets are described in more detail here.

storageOS

A `storageos` volume allows an existing StorageOS volume to be mounted 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 via 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.

```
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` name
        volumeName: redis-vol01
        fsType: ext4
```

For more information including Dynamic Provisioning and Persistent Volume Claims, please see the StorageOS examples.

vsphereVolume

> **Note:** Prerequisite: Kubernetes with vSphere Cloud Provider configured. For cloudprovider configuration please refer 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.

> **Caution:** You must create VMDK using one of the following method before using with 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 Example configuration

```
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
```

More examples can be found here.

Using subPath

Sometimes, it is useful to share one volume for multiple uses in a single Pod. The `volumeMounts.subPath` property can be used to specify a sub-path inside the referenced volume instead of its root.

Here is an example of a Pod with a LAMP stack (Linux Apache Mysql PHP) using a single, shared volume. The HTML contents are mapped to its `html` folder, and the databases will be stored in its `mysql` folder:

```
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 v1.11` alpha

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

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

`subPath` directory names can also be constructed from Downward API environment variables. Before you use this feature, you must enable the

`VolumeSubpathEnvExpansion` feature gate.

In this example, a Pod uses `subPath` to create a directory `pod1` within the hostPath volume `/var/log/pods`, using the pod name from the Downward API. 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/h
    volumeMounts:
    - name: workdir1
      mountPath: /logs
      subPath: $(POD_NAME)
  restartPolicy: Never
  volumes:
  - name: workdir1
    hostPath:
      path: /var/log/pods
```

Resources

The storage media (Disk, SSD, etc.) 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.

In the future, we expect that `emptyDir` and `hostPath` volumes will be able to request a certain amount of space using a resource specification, and to select the type of media to use, for clusters that have several media types.

Out-of-Tree Volume Plugins

The Out-of-tree volume plugins include the Container Storage Interface (CSI) and Flexvolume. They enable storage vendors to create custom storage plugins without adding them to the Kubernetes repository.

Before the introduction of CSI and Flexvolume, all volume plugins (like volume types listed above) were "in-tree" meaning they were built, linked, compiled, and shipped with the core Kubernetes binaries and extend the core Kubernetes API. 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 this FAQ.

CSI

FEATURE STATE: `Kubernetes v1.10` beta

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

- The version names contain beta (e.g. v2beta3).
- Code is well tested. Enabling the feature is considered safe. Enabled by default.
- Support for the overall feature will not be dropped, though details may change.
- The schema and/or semantics of objects may change in incompatible ways in a subsequent beta or stable release. When this happens, we will provide instructions for migrating to the next version. This may require deleting, editing, and re-creating API objects. The editing process may require some thought. This may require downtime for applications that rely on the feature.
- Recommended for only non-business-critical uses because of potential for incompatible changes in subsequent releases. If you have multiple clusters that can be upgraded independently, you may be able to relax this restriction.
- **Please do try our beta features and give feedback on them! After they exit beta, it may not be practical for us to make more changes.**

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.

CSI support was introduced as alpha in Kubernetes v1.9 and moved to beta in Kubernetes v1.10.

Once a CSI compatible volume driver is deployed on a Kubernetes cluster, users may use the `csi` volume type to attach, mount, etc. the volumes exposed by the CSI driver.

The `csi` volume type does not support direct reference from Pod and may only be referenced in a Pod via a `PersistentVolumeClaim` object.

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_attributes` 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 object 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 object 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 v1.11` alpha

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

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

Starting with version 1.11, CSI introduced support for raw block volumes, which relies on the raw block volume feature that was introduced in a previous version of Kubernetes. This feature will make it possible for vendors with external CSI drivers to implement raw block volumes support in Kubernetes workloads.

CSI block volume support is feature-gated and turned off by default. To run CSI with block volume support enabled, a cluster administrator must enable the feature for each Kubernetes component using the following feature gate flags:

```
--feature-gates=BlockVolume=true,CSIBlockVolume=true
```

Learn how to setup your PV/PVC with raw block volume support.

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. Flexvolume driver binaries must be installed in a pre-defined volume plugin path on each node (and in some cases master).

Pods interact with Flexvolume drivers through the `flexvolume` in-tree plugin. More details can be found here.

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 `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 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

> **Caution:** `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.

Edit This Page

Persistent Volumes

This document describes the current state of `PersistentVolumes` in Kubernetes. Familiarity with volumes is suggested.

- Introduction
- Lifecycle of a volume and claim
- Types of Persistent Volumes
- Persistent Volumes
- PersistentVolumeClaims
- Claims As Volumes
- Raw Block Volume Support
- Volume Snapshot and Restore Volume from Snapshot Support
- Writing Portable Configuration

Introduction

Managing storage is a distinct problem from managing compute. 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. 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., can be mounted once read/write or many times read-only).

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.

Please 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 matches 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 in order 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, please check kube-apiserver documentation.

Binding

A user creates, or has already created in the case of dynamic provisioning, 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.

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 which 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` in their Pod's volumes block. See below for syntax details.

Storage Object in Use Protection

The purpose of the Storage Object in Use Protection feature is to ensure that Persistent Volume Claims (PVCs) in active use by a pod and Persistent Volume

(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 the pod status is `Pending` and the pod is assigned to a node or the pod status is `Running`.

When the Storage Object in Use Protection feature is enabled, 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, and also if admin deletes a PV that is bound to a PVC, the PV is not removed immediately. PV removal is postponed until the PV is not bound to a PVC any more.

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=example.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:          Available
Claim:
Reclaim Policy:  Delete
Access Modes:    RWO
Capacity:        1Gi
Message:
Source:
    Type:          HostPath (bare host directory volume)
    Path:          /tmp/data
    HostPathType:
Events:            <none>
```

Reclaiming

When a user is done with their volume, they can delete the PVC objects from the API which 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 here.

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/*
    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.

Expanding Persistent Volumes Claims

FEATURE STATE: `Kubernetes v1.8` alpha

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

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

FEATURE STATE: `Kubernetes v1.11` beta

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

- The version names contain beta (e.g. v2beta3).
- Code is well tested. Enabling the feature is considered safe. Enabled by default.

- Support for the overall feature will not be dropped, though details may change.
- The schema and/or semantics of objects may change in incompatible ways in a subsequent beta or stable release. When this happens, we will provide instructions for migrating to the next version. This may require deleting, editing, and re-creating API objects. The editing process may require some thought. This may require downtime for applications that rely on the feature.
- Recommended for only non-business-critical uses because of potential for incompatible changes in subsequent releases. If you have multiple clusters that can be upgraded independently, you may be able to relax this restriction.
- **Please do try our beta features and give feedback on them! After they exit beta, it may not be practical for us to make more changes.**

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

You can only expand a PVC if its storage class's `allowVolumeExpansion` field is set to true.

```
kind: StorageClass
apiVersion: storage.k8s.io/v1
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.

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 started using the `PersistentVolumeClaim` in ReadWrite mode. Therefore, if a pod or deployment is using a volume and you want to expand it, you need to delete or recreate the pod after the volume has been expanded by the cloud provider in the controller-manager. You can check the status of resize operation by running the `kubectl describe pvc` command:

```
kubectl describe pvc <pvc_name>
```

If the `PersistentVolumeClaim` has the status `FileSystemResizePending`, it is safe to recreate the pod using the PersistentVolumeClaim.

Resizing an in-use PersistentVolumeClaim

FEATURE STATE: `Kubernetes v1.11` alpha

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

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

Expanding in-use PVCs is an alpha feature. To use it, enable the `ExpandInUsePersistentVolumes` feature gate. 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 which uses the PVC before the expansion can complete.

> **Note:** Expanding EBS volumes is a time consuming operation. Also, there is a per-volume quota of one modification every 6 hours.

Types of Persistent Volumes

`PersistentVolume` types are implemented as plugins. Kubernetes currently supports the following plugins:

- GCEPersistentDisk

- AWSElasticBlockStore
- AzureFile
- AzureDisk
- FC (Fibre Channel)
- Flexvolume
- Flocker
- NFS
- iSCSI
- RBD (Ceph Block Device)
- CephFS
- Cinder (OpenStack block storage)
- Glusterfs
- VsphereVolume
- Quobyte Volumes
- HostPath (Single node testing only – local storage is not supported in any way and WILL NOT WORK in a multi-node cluster)
- Portworx Volumes
- ScaleIO Volumes
- StorageOS

Persistent Volumes

Each PV contains a spec and status, which is the specification and status of the volume.

```
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
```

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 v1.9` alpha

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

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

To enable this feature, enable the `BlockVolume` feature gate on the apiserver, controller-manager and the kubelet.

Prior to Kubernetes 1.9, all volume plugins created a filesystem on the persistent volume. Now, you can set the value of `volumeMode` to `raw` to use a raw block device, or `filesystem` to use a filesystem. `filesystem` is the default if the value is omitted. This is an optional API parameter.

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	ReadWriteOnce	ReadOnlyMany	ReadWriteMany
AWSElasticBlockStore		-	-
AzureFile			
AzureDisk		-	-
CephFS			
Cinder		-	-
FC			-
Flexvolume			-
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.

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.

```
kind: PersistentVolumeClaim
apiVersion: v1
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 indicates 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 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.

```
kind: Pod
apiVersion: v1
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 (`ROX`, `RWX`) is only possible within one namespace.

Raw Block Volume Support

FEATURE STATE: `Kubernetes v1.9` alpha

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

- The version names contain alpha (e.g. v1alpha1).
- Might be buggy. Enabling the feature may expose bugs. Disabled by default.

- Support for feature may be dropped at any time without notice.
- The API may change in incompatible ways in a later software release without notice.
- Recommended for use only in short-lived testing clusters, due to increased risk of bugs and lack of long-term support.

To enable support for raw block volumes, enable the `BlockVolume` feature gate on the apiserver, controller-manager and the kubelet.

The following volume plugins support raw block volumes, including dynamic provisioning where applicable.

- AWSElasticBlockStore
- AzureDisk
- FC (Fibre Channel)
- GCEPersistentDisk
- iSCSI
- Local volume
- RBD (Ceph Block Device)

  Note: Only FC and iSCSI volumes supported raw block volumes in Kubernetes 1.9. Support for the additional plugins was added in 1.10.

Persistent Volumes 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: ["50060e801049cfd1"]
    lun: 0
    readOnly: false
```

Persistent Volume Claim 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, we 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

PV volumeMode	PVC volumeMode	Result
unspecified	Block	NO BIND
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 v1.12` alpha

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

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

Volume snapshot feature was added to support CSI Volume Plugins only. For details, see volume snapshots.

To enable support for restoring a volume from a volume snapshot data source, enable the `VolumeSnapshotDataSource` feature gate on the apiserver and controller-manager.

Create Persistent Volume Claim from 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
```

Writing Portable Configuration

If you're writing configuration templates or examples that run on a wide range of clusters and need persistent storage, we recommend that you use the following pattern:

- Do 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, do 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).

Edit This Page

Volume Snapshots

This document describes the current state of `VolumeSnapshots` in Kubernetes. Familiarity with persistent volumes is suggested.

- Introduction
- Lifecycle of a volume snapshot and volume snapshot content
- Volume Snapshot Contents
- VolumeSnapshots

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.

While `VolumeSnapshots` allow a user to consume abstract storage resources, cluster administrators need to be able to offer a variety of `VolumeSnapshotContents` without exposing users to the details of how those volume snapshots should be provisioned. For these needs there is the `VolumeSnapshotClass` resource.

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, the Kubernetes team provides a sidecar helper container for the snapshot controller called `external-snapshotter`. It watches `VolumeSnapshot` objects and triggers `CreateSnapshot` and `DeleteSnapshot` operations against a CSI endpoint.
- 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 `external-snapshotter`.
- The CSI drivers that support volume snapshot will automatically install CRDs defined for the volume snapshots.

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: statically or dynamically.

Static

A cluster administrator creates a number of `VolumeSnapshotContents`. 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 `VolumeSnapshotContents` the administrator created matches a user's `VolumeSnapshot`, the cluster may try to dynamically provision a volume snapshot specially for the `VolumeSnapshot` object. This provisioning is based on `VolumeSnapshotClasses`: the `VolumeSnapshot` must request a volume snapshot class and the administrator must have created and configured that class in order for dynamic provisioning to occur.

Binding

A user creates, or has already created in the case of dynamic provisioning, a `VolumeSnapshot` with a specific amount of storage requested and with certain access modes. A control loop watches for new VolumeSnapshots, finds a matching VolumeSnapshotContent (if possible), and binds them together. If a VolumeSnapshotContent was dynamically provisioned for a new VolumeSnapshot, the loop will always bind that VolumeSnapshotContent to the VolumeSnapshot. Once bound, `VolumeSnapshot` binds are exclusive, regardless of how they were bound. A VolumeSnapshot to VolumeSnapshotContent binding is a one-to-one mapping.

VolumeSnapshots will remain unbound indefinitely if a matching VolumeSnapshotContent does not exist. VolumeSnapshots will be bound as matching VolumeSnapshotContents become available.

Delete

Deletion removes both the `VolumeSnapshotContent` object from the Kubernetes API, as well as the associated storage asset in the external infrastructure.

Volume Snapshot Contents

Each VolumeSnapshotContent contains a spec, which is the specification of the volume snapshot.

```
apiVersion: snapshot.storage.k8s.io/v1alpha1
kind: VolumeSnapshotContent
metadata:
  name: new-snapshot-content-test
spec:
  snapshotClassName: csi-hostpath-snapclass
  source:
    name: pvc-test
    kind: PersistentVolumeClaim
  volumeSnapshotSource:
    csiVolumeSnapshotSource:
      creationTime:    1535478900692119403
      driver:          csi-hostpath
      restoreSize:     10Gi
      snapshotHandle:  7bdd0de3-aaeb-11e8-9aae-0242ac110002
```

Class

A VolumeSnapshotContent can have a class, which is specified by setting the `snapshotClassName` attribute to the name of a VolumeSnapshotClass. A VolumeSnapshotContent of a particular class can only be bound to VolumeSnapshots requesting that class. A VolumeSnapshotContent with no `snapshotClassName` has no class and can only be bound to VolumeSnapshots that request no particular class.

VolumeSnapshots

Each VolumeSnapshot contains a spec and a status, which is the specification and status of the volume snapshot.

```
apiVersion: snapshot.storage.k8s.io/v1alpha1
kind: VolumeSnapshot
metadata:
  name: new-snapshot-test
```

```
spec:
  snapshotClassName: csi-hostpath-snapclass
  source:
    name: pvc-test
    kind: PersistentVolumeClaim
```

Class

A volume snapshot can request a particular class by specifying the name of a VolumeSnapshotClass using the attribute `snapshotClassName`. Only VolumeSnapshotContents of the requested class, ones with the same `snapshotClassName` as the VolumeSnapshot, can be bound to the VolumeSnapshot.

Edit This Page

Storage Classes

This document describes the concept of a StorageClass in Kubernetes. Familiarity with volumes and persistent volumes is suggested.

- Introduction
- The StorageClass Resource
- Parameters

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.

```
kind: StorageClass
apiVersion: storage.k8s.io/v1
metadata:
  name: standard
provisioner: kubernetes.io/aws-ebs
parameters:
  type: gp2
reclaimPolicy: Retain
mountOptions:
  - debug
volumeBindingMode: Immediate
```

Provisioner

Storage classes have a provisioner that determines what volume plugin is used for provisioning PVs. This field must be specified.

Volume Plugin	Internal Provisioner	Config Example
AWSElasticBlockStore		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
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-incubator/external-storage houses a library for writing external provisioners that implements the bulk of the specification plus various community-maintained external provisioners.

For example, NFS doesn't provide an internal provisioner, but an external provisioner can be used. Some external provisioners are listed under the repository kubernetes-incubator/external-storage. There are also cases when 3rd party storage vendors provide their own external provisioner.

Reclaim Policy

Persistent Volumes that are dynamically created by a storage class 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`.

Persistent Volumes that are created manually and managed via a storage class will have whatever reclaim policy they were assigned at creation.

Mount Options

Persistent Volumes that are dynamically created by a storage class 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

FEATURE STATE: `Kubernetes v1.12` beta

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

- The version names contain beta (e.g. v2beta3).
- Code is well tested. Enabling the feature is considered safe. Enabled by default.
- Support for the overall feature will not be dropped, though details may change.
- The schema and/or semantics of objects may change in incompatible ways in a subsequent beta or stable release. When this happens, we will provide instructions for migrating to the next version. This may require deleting,

editing, and re-creating API objects. The editing process may require some thought. This may require downtime for applications that rely on the feature.
- Recommended for only non-business-critical uses because of potential for incompatible changes in subsequent releases. If you have multiple clusters that can be upgraded independently, you may be able to relax this restriction.
- **Please do try our beta features and give feedback on them! After they exit beta, it may not be practical for us to make more changes.**

  Note: This feature requires the `VolumeScheduling` feature gate to be enabled.

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:

- AWSElasticBlockStore
- GCEPersistentDisk
- AzureDisk

The following plugins support `WaitForFirstConsumer` with pre-created PersistentVolume binding:

- All of the above
- Local

Allowed Topologies

FEATURE STATE: `Kubernetes v1.12` beta

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

- The version names contain beta (e.g. v2beta3).
- Code is well tested. Enabling the feature is considered safe. Enabled by default.
- Support for the overall feature will not be dropped, though details may change.
- The schema and/or semantics of objects may change in incompatible ways in a subsequent beta or stable release. When this happens, we will provide instructions for migrating to the next version. This may require deleting, editing, and re-creating API objects. The editing process may require some thought. This may require downtime for applications that rely on the feature.
- Recommended for only non-business-critical uses because of potential for incompatible changes in subsequent releases. If you have multiple clusters that can be upgraded independently, you may be able to relax this restriction.
- **Please do try our beta features and give feedback on them! After they exit beta, it may not be practical for us to make more changes.**

  Note: This feature requires the `VolumeScheduling` feature gate to be enabled.

When a cluster operactor 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.

```
kind: StorageClass
apiVersion: storage.k8s.io/v1
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 `io1`, for the parameter `type`, and the parameter `iopsPerGB` are specific to EBS. When a parameter is omitted, some default is used.

AWS EBS

```
kind: StorageClass
apiVersion: storage.k8s.io/v1
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

```
kind: StorageClass
apiVersion: storage.k8s.io/v1
metadata:
  name: slow
provisioner: kubernetes.io/gce-pd
parameters:
  type: pd-standard
  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.
- `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. In this case, users must use `zones` instead of `zone` to specify the desired replication zones. If exactly two zones are specified, the Regional PD will be provisioned in those zones. If more than two zones are specified, Kubernetes will arbitrarily choose among the specified zones. If the `zones` parameter is omitted, Kubernetes will arbitrarily choose among zones managed by the cluster.

> **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"`, e.g. 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,42982310de6c63381718ccfa6d8`
  This is an optional parameter.
- `gidMin`, `gidMax` : The minimum and maximum value of GID range for the storage class. 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

```
kind: StorageClass
apiVersion: storage.k8s.io/v1
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 1.11` deprecated

  This feature is *deprecated*. For more information on this state, see the Kubernetes Deprecation Policy.

  This internal provisioner of OpenStack is deprecated. Please use the external cloud provider for OpenStack.

vSphere

1. Create a StorageClass with a user specified disk format.

```
kind: StorageClass
apiVersion: storage.k8s.io/v1
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.

```
kind: StorageClass
apiVersion: storage.k8s.io/v1
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 storage class, 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

```
kind: StorageClass
apiVersion: storage.k8s.io/v1
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", e.g. created in this way:

```
kubectl create secret generic ceph-secret --type="kubernetes.io/rbd" \
  --from-literal=key='QVFEQ1pMdFhPUnQrSmhBQUFYaERWNHJsZ3BsMmNjcDR6RFZST0E9PQ==' \
  --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.q. `<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", e.g. created in this way:

```
kubectl create secret generic quobyte-admin-secret \
  --type="kubernetes.io/quobyte" --from-literal=key='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

```
kind: StorageClass
apiVersion: storage.k8s.io/v1
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.

New Azure Disk Storage Class (starting from v1.7.2)

```
kind: StorageClass
apiVersion: storage.k8s.io/v1
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.
- 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

```
kind: StorageClass
apiVersion: storage.k8s.io/v1
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.

During provision, a secret is created for 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`.

Portworx Volume

```
kind: StorageClass
apiVersion: storage.k8s.io/v1
metadata:
  name: portworx-io-priority-high
provisioner: kubernetes.io/portworx-volume
```

```
parameters:
  repl: "1"
  snap_interval:   "70"
  io_priority:  "high"
```

- `fs`: filesystem to be laid out: none/xfs/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 A string is expected here i.e. `"1"` and not `1`.
- `io_priority`: determines whether the volume will be created from higher performance or a lower priority storage high/medium/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. A string is expected here i.e. `"true"` and not `true`.

ScaleIO

```
kind: StorageClass
apiVersion: storage.k8s.io/v1
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

```
kind: StorageClass
apiVersion: storage.k8s.io/v1
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 v1.10` beta

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

- The version names contain beta (e.g. v2beta3).
- Code is well tested. Enabling the feature is considered safe. Enabled by default.
- Support for the overall feature will not be dropped, though details may change.
- The schema and/or semantics of objects may change in incompatible ways in a subsequent beta or stable release. When this happens, we will provide instructions for migrating to the next version. This may require deleting, editing, and re-creating API objects. The editing process may require some thought. This may require downtime for applications that rely on the feature.
- Recommended for only non-business-critical uses because of potential for incompatible changes in subsequent releases. If you have multiple clusters that can be upgraded independently, you may be able to relax this restriction.
- **Please do try our beta features and give feedback on them! After they exit beta, it may not be practical for us to make more changes.**

```
kind: StorageClass
apiVersion: storage.k8s.io/v1
metadata:
  name: local-storage
provisioner: kubernetes.io/no-provisioner
volumeBindingMode: WaitForFirstConsumer
```

Local volumes do not support dynamic provisioning yet, 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.

Edit This Page

Volume Snapshot Classes

This document describes the concept of `VolumeSnapshotClass` in Kubernetes. Familiarity with volume snapshots and storage classes is suggested.

- Introduction
- The VolumeSnapshotClass Resource
- Parameters

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 `snapshotter` 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.

Administrators can specify a default `VolumeSnapshotClass` just for VolumeSnapshots that don't request any particular class to bind to.

```
apiVersion: snapshot.storage.k8s.io/v1alpha1
kind: VolumeSnapshotClass
metadata:
  name: csi-hostpath-snapclass
snapshotter: csi-hostpath
parameters:
```

Snapshotter

Volume snapshot classes have a snapshotter that determines what CSI volume plugin is used for provisioning VolumeSnapshots. This field must be specified.

Parameters

Volume snapshot classes have parameters that describe volume snapshots belonging to the volume snapshot class. Different parameters may be accepted depending on the `snapshotter`.

Edit This Page

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
- Enabling Dynamic Provisioning
- Using Dynamic Provisioning
- Defaulting Behavior
- Topology Awareness

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 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.

Edit This Page

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
- Custom limits
- Dynamic volume limits

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.

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 v1.12` beta

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

- The version names contain beta (e.g. v2beta3).
- Code is well tested. Enabling the feature is considered safe. Enabled by default.
- Support for the overall feature will not be dropped, though details may change.
- The schema and/or semantics of objects may change in incompatible ways in a subsequent beta or stable release. When this happens, we will provide instructions for migrating to the next version. This may require deleting, editing, and re-creating API objects. The editing process may require some thought. This may require downtime for applications that rely on the feature.

- Recommended for only non-business-critical uses because of potential for incompatible changes in subsequent releases. If you have multiple clusters that can be upgraded independently, you may be able to relax this restriction.
- **Please do try our beta features and give feedback on them! After they exit beta, it may not be practical for us to make more changes.**

Kubernetes 1.11 introduced support for dynamic volume limits based on Node type as an Alpha feature. In Kubernetes 1.12 this feature is graduating to Beta and will be enabled by default.

Dynamic volume limits is supported for following volume types.

- Amazon EBS
- Google Persistent Disk
- Azure Disk
- CSI

When the dynamic volume limits feature is enabled, Kubernetes automatically determines the Node type and enforces the appropriate number of attachable volumes for the node. For example:

- On Google Compute Engine, up to 128 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.
- For CSI, any driver that advertises volume attach limits via CSI specs will have those limits available as the Node's allocatable property and the Scheduler will not schedule Pods with volumes on any Node that is already at its capacity. Refer to the CSI specs for more details.

Edit This Page

Pod Security Policies

FEATURE STATE: `Kubernetes v1.12` beta

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

- The version names contain beta (e.g. v2beta3).

- Code is well tested. Enabling the feature is considered safe. Enabled by default.
- Support for the overall feature will not be dropped, though details may change.
- The schema and/or semantics of objects may change in incompatible ways in a subsequent beta or stable release. When this happens, we will provide instructions for migrating to the next version. This may require deleting, editing, and re-creating API objects. The editing process may require some thought. This may require downtime for applications that rely on the feature.
- Recommended for only non-business-critical uses because of potential for incompatible changes in subsequent releases. If you have multiple clusters that can be upgraded independently, you may be able to relax this restriction.
- **Please do try our beta features and give feedback on them! After they exit beta, it may not be practical for us to make more changes.**

Pod Security Policies enable fine-grained authorization of pod creation and updates.

- What is a Pod Security Policy?
- Enabling Pod Security Policies
- Authorizing Policies
- Policy Order
- Example
- Policy Reference

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`
White list of 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`, `supplementalGroups`
Restricting escalation to root privileges	`allowPrivilegeEscalation`, `defaultAllowPrivileg`
Linux capabilities	`defaultAddCapabilities`, `requiredDropCapabiliti`
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	annotations

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 the 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:

```
kind: ClusterRole
apiVersion: rbac.authorization.k8s.io/v1
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):

```
kind: ClusterRoleBinding
apiVersion: rbac.authorization.k8s.io/v1
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. Otherwise requests would bypass authentication and authorization modules, all PodSecurityPolicy objects would be allowed, and users would be able 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 in the following order:

1. If any policies successfully validate the pod without altering it, they are used.
2. If it is a pod creation request, then the first valid policy in alphabetical order is used.
3. Otherwise, if it is a pod update request, an error is returned, because pod mutations are disallowed during update operations.

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=ps
```

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.

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
Error from server (Forbidden): error when creating "STDIN": pods "pause" is forbidden: unabl
```

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
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
Error from server (Forbidden): error when creating "STDIN": pods "privileged" is forbidden:
```

Delete the pod before moving on:

```
kubectl-user delete pod pause
```

Run another pod

Let's try that again, slightly differently:

```
kubectl-user run 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                TYP
1m         2m          15        pause-7774d79b5   ReplicaSet                            War
```

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
^C
```

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 restricted 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'
    apparmor.security.beta.kubernetes.io/allowedProfileNames: 'runtime/default'
    seccomp.security.alpha.kubernetes.io/defaultProfileName:  'docker/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
```

`policy/restricted-psp.yaml`

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 whitelist 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.

AllowedHostPaths - See Volumes and file systems.

Volumes and file systems

Volumes - Provides a whitelist 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

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 `FSGroups` 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 whitelist 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 whitelist 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: extensions/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. 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.
- *MustRunAsNonRoot* - Requires that the pod be submitted with a non-zero `runAsUser` or have the `USER` directive defined (using a numeric GID) in the image. No default provided. Setting `allowPrivilegeEscalation=false` is strongly recommended with this strategy.
- *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 whitelist of capabilities that may 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 whitelist 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 in tact with no modifications. This is denoted as the string `Unmasked`.

AppArmor

Controlled via annotations on the PodSecurityPolicy. Refer to the AppArmor documentation.

Seccomp

The use of seccomp profiles in pods can be controlled via annotations on the PodSecurityPolicy. Seccomp is an alpha feature in Kubernetes.

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.
- `docker/default` - The Docker default seccomp profile is used.
- `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.

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

Controlled via annotations on the PodSecurityPolicy. Refer to the Sysctl documentation.

Edit This Page

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.

- Enabling Resource Quota
- Compute Resource Quota
- Storage Resource Quota
- Object Count Quota
- Quota Scopes
- Requests vs Limits
- Viewing and Setting Quotas
- Quota and Cluster Capacity
- Limit Priority Class consumption by default
- Example
- What's next

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 project.

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.

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 apiserver `--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
`cpu`	Across all pods in a non-terminal state, the sum of CPU requests cannot exceed this valu
`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 val
`memory`	Across all pods in a non-terminal state, the sum of memory requests cannot exceed this
`requests.cpu`	Across all pods in a non-terminal state, the sum of CPU requests cannot exceed this valu
`requests.memory`	Across all pods in a non-terminal state, the sum of memory requests cannot exceed this

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
`requests.storage`	Across all persistent v
`persistentvolumeclaims`	The total number of p
`<storage-class-name>.storageclass.storage.k8s.io/requests.storage`	Across all persistent v
`<storage-class-name>.storageclass.storage.k8s.io/persistentvolumeclaims`	Across all persistent v

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
`requests.ephemeral-storage`	Across all pods in the namespace, the sum of local ephemeral storage reque
`limits.ephemeral-storage`	Across all pods in the namespace, the sum of local ephemeral storage limits

Object Count Quota

The 1.9 release added support to quota all standard namespaced resource types using the following syntax:

- `count/<resource>.<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`
- `count/deployments.extensions`

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 quota 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 may choose to quota jobs to protect against a poorly configured cronjob creating too many jobs in a namespace causing a denial of service.

Prior to the 1.9 release, it was possible to do generic object count quota on a limited set of resources. In addition, it is possible to further constrain quota for particular resources by their type.

The following types are supported:

Resource Name	Description
`configmaps`	The total number of config maps that can exist in the namespace.
`persistentvolumeclaims`	The total number of persistent volume claims that can exist in the namespace.
`pods`	The total number of pods in a non-terminal state that can exist in the namespac
`replicationcontrollers`	The total number of replication controllers that can exist in the namespace.
`resourcequotas`	The total number of resource quotas 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 load balancer that can exist in the namespa
`services.nodeports`	The total number of services of type node port 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 `.spec.activeDeadlineSeconds >= 0`
`NotTerminating`	Match pods where `.spec.activeDeadlineSeconds is nil`
`BestEffort`	Match pods that have best effort quality of service.
`NotBestEffort`	Match pods that do not have best effort quality of service.

The `BestEffort` scope restricts a quota to tracking the following resource: `pods`

The `Terminating`, `NotTerminating`, and `NotBestEffort` scopes restrict a quota to tracking the following resources:

- `cpu`
- `limits.cpu`
- `limits.memory`
- `memory`
- `pods`
- `requests.cpu`
- `requests.memory`

Resource Quota Per PriorityClass

FEATURE STATE: `Kubernetes 1.12` beta

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

- The version names contain beta (e.g. v2beta3).
- Code is well tested. Enabling the feature is considered safe. Enabled by default.
- Support for the overall feature will not be dropped, though details may change.

- The schema and/or semantics of objects may change in incompatible ways in a subsequent beta or stable release. When this happens, we will provide instructions for migrating to the next version. This may require deleting, editing, and re-creating API objects. The editing process may require some thought. This may require downtime for applications that rely on the feature.
- Recommended for only non-business-critical uses because of potential for incompatible changes in subsequent releases. If you have multiple clusters that can be upgraded independently, you may be able to relax this restriction.
- **Please do try our beta features and give feedback on them! After they exit beta, it may not be practical for us to make more changes.**

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.

> **Note:** You need to enable the feature gate `ResourceQuotaScopeSelectors`before using resource quotas per PriorityClass.

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
```

`scopeSelector` supports the following values in the `operator` field:

- `In`
- `NotIn`
- `Exist`
- `DoesNotExist`

Requests vs 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:
    pods: "4"
    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"
    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
pods                     0     4
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
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.extensions=2,count/replicasets.extensions
```

```
kubectl run nginx --image=nginx --replicas=2 --namespace=myspace
```

```
kubectl describe quota --namespace=myspace
```

```
Name:                         test
Namespace:                    myspace
Resource                      Used  Hard
--------                      ----  ----
count/deployments.extensions  1     2
count/pods                    2     3
count/replicasets.extensions  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 will be able to restrict usage of certain high priority classes to a limited number of namespaces and not every namespaces 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.k8s.io/v1alpha1
kind: AdmissionConfiguration
plugins:
- name: "ResourceQuota"
  configuration:
    apiVersion: resourcequota.admission.k8s.io/v1beta1
    kind: Configuration
    limitedResources:
    - resource: pods
    matchScopes:
    - operator : In
      scopeName: PriorityClass
      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:
      - operator : In
        scopeName: PriorityClass
        values: ["cluster-services"]
```

See LimitedResources and Quota support for priority class design doc for more information.

Example

See a detailed example for how to use resource quota.

What's next

See ResourceQuota design doc for more information.

Edit This Page

Pod Security Policies

FEATURE STATE: `Kubernetes v1.12` beta

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

- The version names contain beta (e.g. v2beta3).
- Code is well tested. Enabling the feature is considered safe. Enabled by default.
- Support for the overall feature will not be dropped, though details may change.
- The schema and/or semantics of objects may change in incompatible ways in a subsequent beta or stable release. When this happens, we will provide instructions for migrating to the next version. This may require deleting, editing, and re-creating API objects. The editing process may require some thought. This may require downtime for applications that rely on the feature.
- Recommended for only non-business-critical uses because of potential for incompatible changes in subsequent releases. If you have multiple clusters that can be upgraded independently, you may be able to relax this restriction.
- **Please do try our beta features and give feedback on them! After they exit beta, it may not be practical for us to make more changes.**

Pod Security Policies enable fine-grained authorization of pod creation and updates.

- What is a Pod Security Policy?
- Enabling Pod Security Policies
- Authorizing Policies
- Policy Order

- Example
- Policy Reference

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`
White list of Flexvolume drivers	`allowedFlexVolumes`
Allocating an FSGroup that owns the pod's volumes	`fsGroup`
Requiring the use of a read only root file system	`readOnlyRootFilesystem`
The user and group IDs of the container	`runAsUser`, `supplementalGroups`
Restricting escalation to root privileges	`allowPrivilegeEscalation`, `defaultAllowPrivileg`
Linux capabilities	`defaultAddCapabilities`, `requiredDropCapabiliti`
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	annotations

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 the 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:

```
kind: ClusterRole
apiVersion: rbac.authorization.k8s.io/v1
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):

```
kind: ClusterRoleBinding
apiVersion: rbac.authorization.k8s.io/v1
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. Otherwise requests would bypass authentication and authorization modules, all PodSecurityPolicy objects would be allowed, and users would be able 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 in the following order:

1. If any policies successfully validate the pod without altering it, they are used.
2. If it is a pod creation request, then the first valid policy in alphabetical order is used.
3. Otherwise, if it is a pod update request, an error is returned, because pod mutations are disallowed during update operations.

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=ps
```

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.

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
Error from server (Forbidden): error when creating "STDIN": pods "pause" is forbidden: unabl
```

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
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
Error from server (Forbidden): error when creating "STDIN": pods "privileged" is forbidden:
```

Delete the pod before moving on:

```
kubectl-user delete pod pause
```

Run another pod

Let's try that again, slightly differently:

```
kubectl-user run 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                TYP
1m         2m          15        pause-7774d79b5   ReplicaSet                            War
```

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
^C
```

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 restricted 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'
    apparmor.security.beta.kubernetes.io/allowedProfileNames: 'runtime/default'
    seccomp.security.alpha.kubernetes.io/defaultProfileName:  'docker/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
```

`policy/restricted-psp.yaml`

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 whitelist 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.

AllowedHostPaths - See Volumes and file systems.

Volumes and file systems

Volumes - Provides a whitelist 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

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 `FSGroups` 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 whitelist 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 whitelist 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: extensions/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. 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.
- *MustRunAsNonRoot* - Requires that the pod be submitted with a non-zero `runAsUser` or have the `USER` directive defined (using a numeric GID) in the image. No default provided. Setting `allowPrivilegeEscalation=false` is strongly recommended with this strategy.
- *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 whitelist of capabilities that may 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 whitelist 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 in tact with no modifications. This is denoted as the string `Unmasked`.

AppArmor

Controlled via annotations on the PodSecurityPolicy. Refer to the AppArmor documentation.

Seccomp

The use of seccomp profiles in pods can be controlled via annotations on the PodSecurityPolicy. Seccomp is an alpha feature in Kubernetes.

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.
- `docker/default` - The Docker default seccomp profile is used.
- `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.

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

Controlled via annotations on the PodSecurityPolicy. Refer to the Sysctl documentation.